

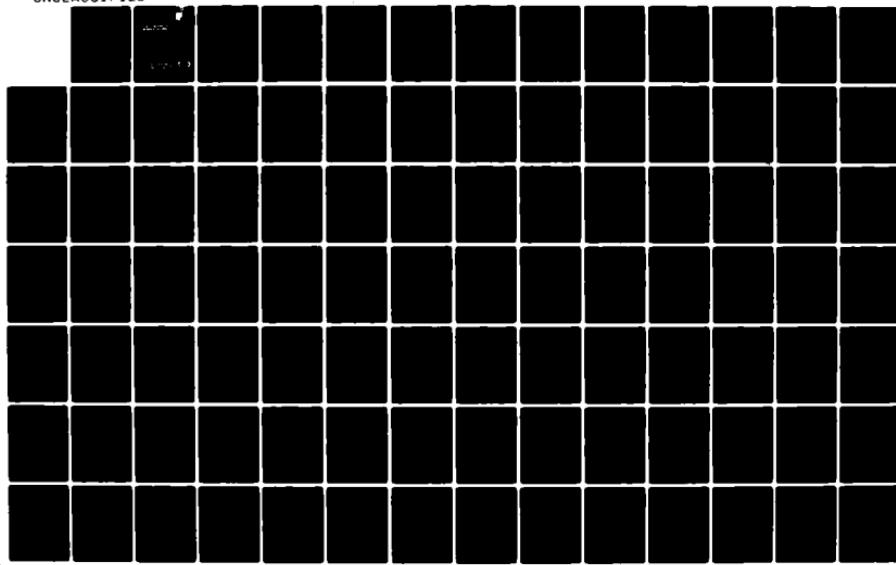
AD-A126 101 TOTAL SYSTEM DESIGN (TSD) METHODOLOGY ASSESSMENT(U)
WASHINGTON UNIV SEATTLE DEPT OF COMPUTER SCIENCE
G ROMAN ET AL. JAN 83 RADC-TR-82-331 F30602-80-C-0284

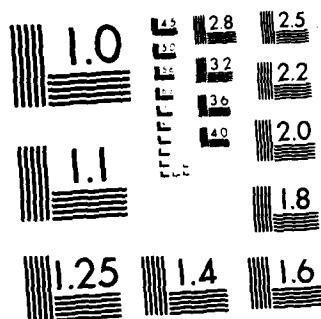
1/4

F/G 9/2

NL

UNCLASSIFIED





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963

ADA 126101

RADC-TR-82-331
Final Technical Report
January 1983



TOTAL SYSTEM DESIGN (TSD) METHODOLOGY ASSESSMENT

Washington University

**Gruia-Catalin Roman, M. J. Stucki, R. K. Israel, W. E. Ball,
W. D. Gillett, S. V. Pollack and J. T. Love**

APPROVED FOR PUBLIC RELEASE: DISTRIBUTION UNLIMITED

**ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441**

**DTIC
ELECTED
MAR 28 1983
S D
D**

83 03 28 040

DTIC FILE COPY

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-82-331 has been reviewed and is approved for publication.

APPROVED:



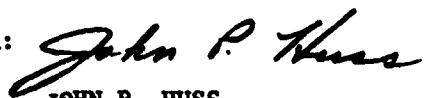
ROGER B. PANARA
Project Engineer

APPROVED:



ALAN R. BARNUM, Assistant Chief
Command & Control Division

FOR THE COMMANDER:


JOHN P. HUSS
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COEE) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document requires that it be returned.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER RADC-TR-82-331	2. GOVT ACCESSION NO. AD-4126101	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) TOTAL SYSTEM DESIGN (TSD) METHODOLOGY ASSESSMENT	5. TYPE OF REPORT & PERIOD COVERED Final Technical Report Sep 80 - Sep 82	
	6. PERFORMING ORG. REPORT NUMBER N/A	
7. AUTHOR(s) Gruia-Catalin Roman M.J. Stucki R.K. Israel	W.E. Ball W.D. Gillett S.V. Pollack	8. CONTRACT OR GRANT NUMBER(s) F30602-80-C-0284
9. PERFORMING ORGANIZATION NAME AND ADDRESS Washington University (Computer Science Dept.) Box 1045 St Louis MO 63130	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 63701B 32050325	
11. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (COEE) Griffiss AFB NY 13441	12. REPORT DATE January 1983	
	13. NUMBER OF PAGES 382	
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Same	15. SECURITY CLASS. (of this report) UNCLASSIFIED	
	16a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report) Same		
18. SUPPLEMENTARY NOTES RADC Project Engineer: Roger B. Panara (COEE)		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) Formal Design Methodologies Hardware/Software/Firmware Trade-offs		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) The report is concerned with the nature of and strategies involved in the identification of the hardware/software mix that comprises a distributed system. Current state-of-the-art in the areas of software engineering and hardware design/selection is consolidated under the unifying umbrella of a methodological framework called the Total System Design (TSD) Framework. A distributed system design methodology, called the TSD Methodology, is proposed and illustrated in the context provided by two applications; (over)		

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 68 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

ballistic missile defense and cartographic database. The methodology is derived from the framework but its scope is limited to the design activities that establish the overall architecture of the system and lead to the selection of a particular hardware/software mix which may include custom, customized, and off-the-shelf components. Plans for the development of a TSD Facility are also outlined in the report. The facility would make available an integrated set of tools for use in conjunction with the TSD Methodology and would provide a milieu for the development, evaluation and productive use of TSD technology.

The report also includes a guidebook describing the TSD concept in layman terminology and several appendixes treating a variety of related topics: (1) an approach for reducing ambiguities in methodology definitions; (2) a formal treatment of distributed systems design; (3) a rigorous approach to developing formal system requirements; (4) a proposal for a distributed systems specification language; and (5) an assessment based on the concepts of the TSD Facility of the plans for a Modern Programming Environment at the Defense Mapping Agency.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

TSD METHODOLOGY ASSESSMENT

TABLE OF CONTENTS

1. Executive Summary	1
1.1 Background	1
1.2 Objectives	3
1.3 Definition of Terms	4
1.4 Summary of Results	6
1.5 Recommendations	7
1.6 Report Summary	14
2. TSD Framework Consolidation	23
2.1 Introduction	23
2.2 TSD Framework Definition	25
2.3 Stages in the TSD Framework	39
2.3.1 Problem Definition Stage	39
2.3.2 System Design Stage	50
2.3.3 Software Design Stage	64
2.3.4 Machine Design Stage	79
2.3.5 Circuit Design Stage	89
2.3.6 Firmware Design Stage	92
2.4 Hardware/Software Trade-offs	108
2.5 System Life-Cycle Issues and the TSD Framework	110
2.5.1 Development	110
2.5.2 Analysis	111
2.5.3 Enhancement	112
2.5.4 Maintenance	114
2.6 Conclusions	116

Accession Per	
NTIS GRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A	



3. Assessing the Family of TSD Methodologies	117
3.1 Introduction	117
3.2 Characterization of Three Classes of DoD Systems	120
3.2.1 Embedded Systems	121
3.2.2 Information Processing Systems	122
3.2.3 Command, Control and Communication Systems	123
3.3 System Design Needs at DMA	125
3.4 TSD View of Distributed Systems Design	128
3.4.1 Introduction	128
3.4.2 Informal Definition of System Requirements	129
3.4.3 Informal Definition of Processing Model	130
3.4.4 Informal Definition of Hardware/Software Requirements	133
3.4.5 Methodology Outline	134
3.4.6 Formalization of the Design Strategy	143
3.5 Distributed Real-Time Systems Illustration	148
3.5.1 Introduction	148
3.5.2 Ballistic Missile Defense Systems	150
3.5.3 TSD Methodology Applicability	158
3.6 Distributed Data Processing Systems Illustration	161
3.6.1 Introduction	161
3.6.2 Applying the TSD Methodology	163
3.6.3 An Example - Digital Land Mass System (DLMS)	166
3.6.4 A Design - DLMS	168
3.6.5 Conclusions	172
4. TSD Facility Development Master Plan	181
4.1 Introduction	181
4.2 Background	188
4.3 Proposal for a TSD System Family	197
4.4 Recommendations for Facility Development	205
4.5 TSD Facility and System Design for DMA	211

Appendix A: Annotated Bibliography	213
Appendix B: Glossary of Terms	223
Appendix C: TSD Methodology Guidebook	231
Appendix D: On Reducing Ambiguities in Methodology Definitions	279
Appendix E: A Formal Treatment of Distributed Systems Design	295
Appendix F: A Rigorous Approach to Building Formal System Requirements	317
Appendix G: Functional Specification of Distributed Systems	331
Appendix H: Modern Programming Environment Assessment	351

1. EXECUTIVE SUMMARY

1.1 BACKGROUND

Advances in the underlying technology and increased demands from a growing range of highly sophisticated application areas are drastically affecting the complexity of systems development. (A system is defined here as a hardware/software or H/S aggregate supporting a given application.) Technological difficulties stem from an increase in the number of design alternatives and related changes in the nature of the systems being considered. The widespread availability of microprocessors has brought about an increased awareness of the role played by H/S trade-offs in systems development. Successful experimentation with unconventional machine architectures has identified the need for careful consideration of the relationship between problem domain characteristics and the architectural features of the support system. Finally, VLSI technology has placed within the designer's reach specialized high performance (off-the-shelf and custom) devices, while requiring a completely new approach to algorithm design that stresses communication cost minimization, simple interconnection topology, and parallelism.

This new technological climate presents DMA as well as all DoD organizations with great opportunities and new challenges in the area of systems design. The performance of existing systems may be increased through enhancements that take advantage of the new technology. Furthermore, systems of unprecedented sophistication can be conceived in response to the ever growing needs of the national defense. The promise of great achievements, however, is postulated on the premise that this new technology may be used effectively. Effective technology utilization can be realized only by employing appropriate methodologies and design aids.

Early recognition of these trends by Rome Air Development Center (RADC) has been marked by a series of related research and development activities whose starting point was the Total System Design (TSD) concept. It envisions system design as taking place in a support environment consisting of a family of design methodologies and a collection of associated design aids. Moreover, the TSD concept also presumes the ability to easily explore the space of design alternatives every step of the way, and to make rational decisions based primarily on solid technical reasons. The notion of avoiding premature commitments to particular design solutions, such as the a priori selection of specific hardware, is another key component of the concept and one of the motivating factors behind its inception.

DMA involvement in the TSD research and development activities came about due to the realization that the establishment of a TSD Facility at RADC, an important DMA service laboratory, would enable DMA contractors to reduce system development costs while enhancing the quality of the systems being built for DMA. Because the DMA production capabilities depend to a significant degree upon the quality of the computer based systems available at its two production centers, the decision to support the TSD efforts represents a major step toward preparing the organization to meet its future operational needs.

The fact that software development work internal to DMA could also benefit from the existence of a TSD Facility (through importation of proven design and management tools) has been realized only recently. During the Seventies, TSD efforts focused on refinement and evaluation of the concept, and on investigation of the feasibility of the development and integration of the required design aids within a TSD Facility. The present version of the facility is known as the System Architecture Evaluation Facility (SAEF) and is generally viewed as serving a dual purpose. On one hand, it supports some of the aids envisioned under the umbrella of the TSD concept while, on the other, SAEF also provides an experimentation laboratory for research into advanced hardware architectures.

The current SAEF configuration (still under development) stresses the use of powerful emulation engines aimed at providing rapid implementation and analysis of alternate hardware configurations, a capability which establishes SAEF as an effective environment for the development and maintenance of embedded computer systems. Single processor emulation is carried out on a high-speed microprogrammable machine already installed at RADC, the Nanodata QM-1, which may be used both in stand alone mode and as shared resource through a DEC-20. The DEC-20, which is connected to the ARPA net, makes the facility available to distant users. Emulation of distributed systems is anticipated to take place primarily on a Multiple Microprocessor System (MMS) which is still in the design stage. The interface between the MMS and the other components of SAEF is to be done via a bus shared with the QM-1 and an already existing STARAN S-1000 associative processor. Accessible from the ARPA net through an H6180 MULTICS system, STARAN is used as an aid in evaluating single-instruction, multiple-data stream (SIMD) architectures.

Attempts to apply existing TSD technology on DMA type problems have revealed, however, the need to broaden the initial scope of the investigation to include:

- the reevaluation of the system life-cycle definition in view of recent changes in the nature of systems (e.g., distribution of both data and processing) and in the relation between hardware and software;
- the identification of the role played by H/S trade-offs in the system life-cycle;
- the development of distributed systems design methodologies that approach the selection of an appropriate H/S mix in a systematic and objective manner;
- the reevaluation of the TSD Facility definition and plans in light of the growing realization that design environments (e.g., Ada) hold the key to productivity and quality increases in the system design area, i.e., tool integration is as important as tool availability.

These issues are considered in this assessment of the current state of the TSD family of methodologies and of the TSD Facility.

1.2 OBJECTIVES

Consolidation of past accomplishments, assessment of the TSD role in the design of DoD/DMA systems, planning for future efforts, and dissemination of the current state of the TSD based technology to its intended beneficiaries are the four principal goals of this assessment study.

TSD CONCEPT CONSOLIDATION.

The consolidation reflects the current perception of the TSD concept. These views are the result of a maturation process stimulated by technological changes that have occurred in the last decade, and by valuable experience gained on TSD research and development projects. The aim is to determine, from a TSD perspective, the fundamental nature of the decision processes involved in system design and to formalize them so that they become the basis for a rigorous system design approach. Of special concern is the expansion of the present understanding of the dynamics of H/S trade-offs. The result of the consolidation is meant to benefit both researchers and technology development planners.

TSD TECHNOLOGY ASSESSMENT.

The most important criterion used in assessing the TSD technology is its effectiveness in the development, analysis, enhancement, and maintenance of those systems that are most often encountered in DoD/DMA applications. Three such applications are considered in this study, and an evaluation is carried out in order to establish the degree to which they could be supported by existing or postulated TSD-based approaches. The TSD's practical significance is measured by the extent to which such approaches could lead to definite cost reductions and quality improvements.

TSD TECHNOLOGY TRANSFER INTO PRODUCTION AND FUTURE R&D PLANS.

Another key objective of this investigation is the generation of recommendations for plans to accomplish the transfer of the TSD technology into the production environment, and the establishment of future research and development directions for the TSD efforts as a whole. DoD/DMA priorities, previous work, anticipated technological trends, and the more refined and comprehensive nature of the newly consolidated view of the TSD concept are all factors that impact the planning process.

TSD TECHNOLOGY DISSEMINATION.

The rationale behind the dissemination of the TSD technology is to be found in a commitment to the establishment of effective production environments. As such, the development of materials that introduce the community of potential users to this technology is considered to be an important and necessary by-product of this study. The materials are intended to increase the visibility and the utilization of the TSD technology within DoD/DMA. While this offers the benefits of further comprehensive evaluation of the TSD technology within a real production effort, it also creates the opportunity for the exploitation of existing TSD design aids and methodologies.

1.3 DEFINITION OF TERMS

Understanding the thesis of this report requires a good grasp of four fundamental concepts: methodological framework (henceforth called framework), methodology, computer-aided design system and design facility. The definition of these terms and their relevance to TSD are reviewed here in preparation for subsequent sections, where more detailed discussions are to be found.

METHODOLOGICAL FRAMEWORK.

A framework represents a high level non-procedural description of some general problem solving approach. More specifically, it identifies:

- (1) a set of subproblems whose solutions lead to the solution of the target problem; and
- (2) fundamental relationships among the subproblems, without regard to the manner in which one arrives at their solutions.

The framework has the ability to relate the nature of the problem and the essence of its solution without telling HOW, but rather WHAT is involved in solving it.

METHODOLOGY.

In contrast with the framework, a methodology prescribes a particular mode of procedure to be followed in solving a given problem. While the objective of a framework is to characterize a class of feasible solutions by abstracting over a family of methodologies, the goal of a methodology is to define an effective solution for the problem at hand. Effectiveness is achieved by exploiting particular features of the problem or environment through the use of specific techniques or classes of techniques. In the latter case, rules for selecting the most appropriate technique from the class of usable techniques are an integral part of the methodology. As a direct result of the emphasis placed on effectiveness, methodologies are largely problem and environment dependent.

COMPUTER-AIDED DESIGN SYSTEM.

A computer-aided design system provides the designer with an integrated set of tools aimed at increasing his/her productivity through the automation of difficult or time consuming tasks. Most often, the tool set directly supports a class of related design methodologies and the management of projects that use the respective methodologies.

DESIGN FACILITY.

A facility is defined as the means of support (i.e., resources) available at some location for use in the application of certain methodologies to various problems. These resources include people, computer-aided design systems, documentation, tools, physical plant, etc., and they are established for the purpose of enhancing design productivity. Consequently, the resources that make up a particular facility tend to be centered around a specific methodology.

In light of the definitions above, the TSD Framework is a framework which assumes systems design to be its problem domain, postulates successful design on the availability of a disciplined design strategy, and acknowledges:

- (1) the H/S dualism rather than dichotomy,
- (2) the need for a formal H/S trade-offs strategy,
- (3) the advantages of systematic error detection,
- (4) the importance of step-by-step performance evaluation,
- (5) the need for proper evaluation of human interfaces.

Thus, the TSD Framework captures the very essence of the TSD concept. Moreover, the TSD Framework specifies the basic characteristics of an entire family of TSD Methodologies to be supported by a computer-aided design system, called TSD System, incorporated in a powerful TSD Facility. The facility is aimed at providing assistance throughout the entire system life cycle, from development to subsequent analysis, enhancement, and maintenance.

1.4 SUMMARY OF RESULTS

The four key objectives of the study correlate strongly with the levels of abstraction identified in the previous section. The consolidation corresponds to the development of the TSD Framework. The assessment involves an evaluation of several TSD Methodologies with respect to their effectiveness in three critical DoD application areas. The planning consists of a review of the current state of the TSD System and Facility and a determination, in light of the earlier assessment, of a suitable course for the future. Finally, a user-oriented TSD Guidebook is created in order to meet the dissemination objective. The main results of the study are reviewed below.

- The TSD Framework represents a redefinition of the system life-cycle -- systems are treated as H/S aggregates and the life-cycle definitions for hardware and software are brought under a unified umbrella.
- The dynamics of the H/S trade-offs have been identified and their role in the system development life-cycle has been established.
- An approach for the development of system design methodologies from the framework definition has been defined and illustrated.
- A distributed systems design methodology that uses the system requirements to generate hardware and software requirements for the system has been proposed and illustrated for a real-time and a data processing system.
- High level requirements and design structure for the TSD System, the computer aided design system at the center of the TSD Facility, have been developed and have been used in the preparation of a TSD Facility master plan.
- A methodology definition language having the potential to be used for configuration control in computer aided design systems and for project planning has been proposed and illustrated.
- A formal characterization of distributed systems design has been developed in order to establish the requirements definition for specification languages needed in system design.
- A distributed systems design specification language meeting some of the requirements identified in the formal model has been developed and illustrated -- its distinguishing feature is the designer's freedom to define arbitrary communication protocols among concurrent processes.
- The formal model has been used also in the development of a systematic approach to building formal system requirements.
- An assessment of the plans for a Modern Programming Environment at DMA based on the concepts of the TSD Facility.

1.5 RECOMMENDATIONS

The recommendations of this study fall into two categories. The first group deals with the master plan for establishing a TSD Facility. The second addresses the issue of how DMA could take advantage of the TSD technology and methodologies in the interim period during which a TSD Facility is not available and as part of its effort to establish a modern programming environment.

TSD FACILITY DEVELOPMENT MASTER PLAN

The following explicit objectives have determined the nature of the TSD Facility master plan:

- low development cost;
- speedy development;
- limited risk;
- early availability to potential users;
- ability to respond to immediate design needs without compromising the long range requirements;
- smooth growth in capability and range of applicability;
- compatibility with other related DoD efforts (e.g., SAEF, Ada);
- strong interaction between R&D and production efforts.

Because the development of a design facility is generally a high risk and high cost proposition, the strategy adopted in the master plan is to minimize new tool development and focus on integrating off-the-shelf components to the greatest possible extent. While the command language, database view and core tools which characterize the central part of the TSD Environment could be assembled together from existing components, the lack of application specific tools could make it difficult to attract potential users of the prototype TSD Facility. This may be avoided if an already successful existing facility could be used to supply the application oriented tools. SAEF has been selected to meet this objective. This particular choice has several other advantages. It employs a facility which is compatible with the general TSD Concept, it provides continuity to the entire TSD program, it addresses a class of users who feel most acutely the need for a design facility (for embedded systems), and it promises immediate and high payoffs.

The result of these and other considerations is a plan which consists of three concurrent efforts which gradually merge into one. The main stream deals with the selection and integration of the TSD Facility components. The other two focus, respectively, on increasing the effectiveness of the application specific tools through enhancements to the SAEF and on providing the technical support needed for long range planning through the development and evaluation of new TSD Methodologies.

The development and evaluation of new TSD Methodologies is meant to have little or no impact on the near term version of the TSD Facility. The objective is to assist in the later evaluation and subsequent enhancements of the TSD Facility available at the end of this planning period. This is to be accomplished by developing tools to be incorporated in subsequent versions of the facility and methodologies that define the

manner in which such tools should be used in various application areas. Because effective methodologies are application dependent, the plan suggests work to be concentrated only on a few application areas of special significance within DoD. Corresponding independent refinements of the TSD Methodologies should be produced for each selected area. Following the methodology development, empirical evaluations on real-life moderately sized projects should be carried out. The experience should be used to further refine and tune the methodologies to the needs of the respective application areas. The development of specification languages and analysis techniques should be centered around mechanizing some of the activities involved in applying the methodologies. This is the point where some integration between the intentionally independent undertakings ought to take place. The level of effort required by this particular stream of the master plan depends upon the range of applications being chosen. (No more than three areas should be attempted.)

SAEF enhancements are motivated by the desire to make the ultimate facility more attractive to potential users, to build a user community concurrently with the development of the facility, and to establish a realistic base for determining the priority assigned to introducing various core tools. Since it is expected that not all core tools will be available in the prototype facility, those tools that appear to be most needed by the particular community of users ought to be considered first. Furthermore, current understanding of the specification language needs for the system design stage should be used in the design of the next version of the hardware description language used by SAEF. This stream of activities is also independent in nature from the other two.

The main thread of the master plan is concerned with building a TSD System from available components and its integration with the SAEF to form the TSD Facility prototype. The approach is actually consistent with the TSD Methodologies. It starts with the problem definition stage during which a detailed definition of the TSD Environment (only outlined by this study) is generated. Based on the TSD Environment definition a system architecture for the TSD System is developed in a manner which is consistent with the constraint that the proposed architecture must be supported primarily by the resources available in SAEF. (Given the short range nature of the plan hardware procurement ought to be avoided.) Next the binding phase is carried out. It consists of the selection of existing tools required to support various entities of the TSD System and of the definition of custom software needed to integrate them. This activity represents, in the terminology of the TSD Framework, the generation of software requirements. (The hardware is given in this case.) The integration of the tools is carried out in stages. The last one involves placing all acquired tools on the SAEF and thus establishing the TSD Facility prototype. Once some experience with the use of the TSD Facility on several production efforts is accumulated, the facility may be reevaluated and new plans devised for its future.

REFINEMENT OF TSD METHODOLOGIES FOR SEVERAL KEY APPLICATIONS	ENHANCEMENTS TO SAEF CAPABILITIES	DEVELOPMENT OF TSD SYSTEM PROTOTYPE
SELECTION OF KEY APPLICATIONS	STUDY OF POTENTIAL SAEF ENHANCEMENTS	DETAILED DEFINITION OF TSD ENVIRONMENT AND SYSTEM
DEVELOPMENT OF NARROWLY FOCUSED TSD METHODOLOGIES	UPGRADING OF SAEF CAPABILITIES	TSD SYSTEM BINDING (TOOL SELECTION)
EMPIRICAL EVALUATION OF TSD METHODOLOGIES	INTEGRATION OF OFF-THE-SHELF TOOLS	DEVELOPMENT OF CUSTOM COMPONENTS
DEVELOPMENT OF SPECIFICATION LANGUAGES	USE OF ENHANCED SAEF IN PRODUCTION	CONTINUATION OF THE TOOL INTEGRATION
DEVELOPMENT OF ANALYSIS TECHNIQUES	INTEGRATION WITH SAEF CAPABILITIES
		PRODUCTION EXPERIENCE
		TSD FACILITY REVIEW AND PLANNING

TSD FACILITY DEVELOPMENT MASTER PLAN

TSD FACILITY AND SYSTEM DESIGN AT DMA

In order to understand the methodological needs of the DMA, one has to consider the characteristics of the production environment existing at DMA and the nature of the applications with which this organization is involved. In this regard, the following issues seem to have the greatest bearing on the future of system design at DMA.

- The DMA production plan is determined by the mapping, charting, and geodesy (MC&G) defense needs of the many DoD organizations. Changes in the data format, use and collection (quite often unanticipated) bring about increased demands for MC&G products, demands that translate into corresponding enhancements in the systems employed by DMA. Its ability to keep up with future growth indicates a need to employ effective system design methodologies capable of supporting the dynamic evolution experienced by DMA systems.
- While at present most DMA systems could be considered to be of the information processing type, their MC&G nature makes the importation of system design technology somewhat less direct. The following is a list of features unique to geographic data processing:
 - demanding performance constraints not present in other data processing applications;
 - presence of locational attributes;
 - two-dimensional nature of the problem domain;
 - particularly large amount of storage;
 - lack of commercially available systems;
 - government ownership of most existing systems;
 - specialized and expensive input/output devices;
 - dependence upon remote sensing technology.
- All major geographic data processing systems in production today have been developed by some government organization (within or outside the U.S.A.) and have been designed to serve a set of very specific requirements. Consequently, geographic data processing for military purposes receives little attention outside the government and puts DMA in the position of having to develop on its own the system design technology required to maintain and enhance its MC&G production.
- The complexity of the current types of systems is on the rise. The number and volumes of the databases, the workload, and the number and variety of products all experience noticeable growth. Moreover, greater interdependencies between databases and products is anticipated. The ultimate consequence of these trends might be the evolution of a single distributed DMA system, a critical component of the entire organization.
- There is also evidence pointing to a possible new group of systems of the embedded type. Computer controlled devices in use at DMA can be viewed to be in this category already.

Furthermore, any increased future involvement of the organization in the data collection process most certainly is bound to extend DMA related system design efforts into the embedded systems area.

- At a more speculative level, incorporation of DMA systems into larger C3 systems can not be ruled out. Major increases in the data collection rate combined with a need to possess extremely current MC&G products (possibly on-line) may contribute to making this qualitative jump.

The productivity associated with the generation of MC&G products at DMA appears to be related to the quality of the computer based systems being employed, which in turn depends on the effective use of current technology at hardware, software, and system levels. TSD Methodologies hold the potential to assist DMA with many of these system related problems and to provide cohesiveness to long range planning in this area. They extend the ability of the organization to control and manage system development, maintenance, and enhancement. Furthermore, TSD Methodologies promote careful definition of system requirements and more effective use of available technology. In other words, the DMA's strides toward quality, productivity, enhanceability, maintainability, and low system design costs are identical to the basic objectives of the TSD technology.

DMA is in a position to take advantage of the TSD technology in several important ways:

- Contractors could make use of the envisioned TSD Facility on projects involved in the development of DMA systems;
- The TSD technology could be used by DMA contractors, even in the absence of the TSD Facility, particularly in the design of systems which are distributed in nature and involve decisions regarding the selection of a proper hardware/software mix;
- The core tools being developed for the TSD Facility are also needed as part of the DMA modern programming environment (MPE) which is seen as evolving in a TSD Facility specialized in software development;
- The TSD Methodologies may also be used in DMA on certain projects where the relation between software and hardware is important (e.g., the placement of various functions on a locally distributed system) and, thus, could affect DMA software development practices.

The first of the above concerns was discussed in the master plan, and a way to approach the remaining three is outlined below. The direction being suggested here is analogous to that part of the master plan that deals with the refinement of TSD Methodologies. The distinction is not in the basic approach but in the scope and objectives. In the master plan the intent is to define the scope of and to support the long range R&D efforts in the area of distributed system design. Here, the objective is technology transfer from the R&D domain to actual production for the sake

of achieving immediate quality and productivity improvements. As such, the emphasis is not on developing novel design, specification, analysis, and other techniques but rather on adapting already existing techniques for use in some particular application in a manner compatible with the TSD philosophy. It is conceivable that after empirical evaluations via appropriate pilot projects, some limited use of the methodologies on selected projects will become feasible in the near future. The potential impact of such endeavors on the DMA modern programming environment, on its approach to system development, and even on its software development standards should not be underestimated.

IDENTIFICATION OF POTENTIALLY
HIGH PAYOFF AREAS AS
CONTRACTOR AND AS DEVELOPER

REFINEMENT OF THE TSD
METHODOLOGIES WITH RESPECT
TO THE SELECTED AREAS

EMPIRICAL EVALUATION OF
THE METHODOLOGIES ON SEVERAL
SMALL PILOT PROJECTS

EVALUATION OF THE DMA
MODERN PROGRAMMING ENVIRONMENT
WITH RESPECT TO ITS ABILITY
TO SUPPORT THE METHODOLOGIES

ENHANCEMENT OF CURRENT DMA
ENVIRONMENT TOWARD BEING
BETTER PREPARED TO RESPOND
TO FUTURE SYSTEM DESIGN NEEDS

LIMITED USE OF TSD METHODOLOGIES
ON SELECTED DMA PROJECTS

REEVALUATION OF SYSTEM DESIGN
NEEDS AND AVAILABLE TECHNOLOGY
AT DMA

DMA OPPORTUNITIES FOR PRODUCTIVE USE OF TSD TECHNOLOGY

1.6 REPORT SUMMARY

Brief summaries of all major sections and key appendixes of the report are presented here for the reader's convenience.

TSD FRAMEWORK CONSOLIDATION (Section 2).

The section outlines the philosophy, motivation, and significance of the TSD concept and dwells on the structural details of the TSD Framework, stressing its relation to fundamental decision processes that take place during system design. The partitioning of the system functions between hardware and software receives extensive coverage. The relevance of the TSD Framework for system development, analysis, enhancement, and maintenance is also discussed. A strong emphasis is placed on demonstrating the practical advantages derived from the availability of the framework.

The TSD Framework is hierarchical in structure, being composed of stages which are, in turn, composed of phases, which are composed of steps. The stages represent broad design areas such as system design, software design, and hardware design, while the phases represent finer divisions of these design areas. For example, a stage dealing with software design could contain separate phases for software architecture, program design, and coding. The steps represent design activities that go on within the design areas. They include activities such as performance evaluation, functional verification, documentation, and acceptance. The framework describes, in a straightforward manner, the logical organization and the design activities intrinsic to a particular family of design methodologies called TSD Methodologies.

Distinct methodologies emphasize different applications and thus instantiate the phases and steps in different ways. This principle is illustrated on a small example. By selecting a sample application area and by considering its characteristics and their relation to both technology and application environment, a methodology is derived in a systematic manner from the TSD Framework. The approach suggests that, for each application area and organization, methodology development involves a certain degree of "pre-design" in addition to the selection of particular techniques for design, analysis and specification. This fact becomes even more evident in the assessment.

ASSESSING THE FAMILY OF TSD METHODOLOGIES (Section 3).

The TSD assessment starts with an examination of the essential characteristics of the embedded, information processing, and command, control and communication systems. The unique nature of the applications supported by DMA is used to emphasize the dependency between methodologies and the nature of the organization that may employ them. The point is made that future detailed assessments of the TSD technology ought to be carried out not only with respect to a specific class of systems but also with respect to the type of organization that intends to build them. The principal results of the assessment are given below.

- By accomplishing the transition from the TSD Framework to a class of distributed system design methodologies and by describing how one could employ these methodologies on system design projects having characteristics common to a multitude of DoD (including DMA) type systems, the technical feasibility of the TSD Framework is demonstrated.
- The actual use of the concepts and methodological study approaches developed during the consolidation effort (in particular the synthesis of methodologies given a framework and a class of applications) illustrates convincingly the assistance these approaches could provide to methodological research and development.
- The TSD Methodologies are shown to promote a systematic approach to the performance of hardware/software trade-offs thus avoiding the known problem of premature hardware procurement. Future research advances in this area combined with experiments in which these methodologies are applied to real-life systems hold the key to making the employment of these methodologies both practical and profitable in terms of quality and productivity gains.
- Techniques and tools (available or postulated) identified as necessary for productive use of the TSD Methodologies form the starting point for the development of the TSD Facility master plan. It must be noted, however, that there are many other factors that intervene and influence the planning of such a facility in addition to the techniques suggested by the use of one methodology or another.
- Four by-products of the assessment are:
 - a methodology definition language;
 - a formal characterization of the nature of the specification languages involved in system design;
 - a rigorous approach to developing formal system requirements;
 - a distributed system design specification language.

TSD FACILITY DEVELOPMENT MASTER PLAN (Section 4).

Three steps were involved in the development of the TSD Facility master plan. Their respective objectives are discussed below.

- Objective 1: Develop a conceptual model for an integrated set of design tools to support the first three stages of the TSD Framework (Problem Definition, System Design, Software Design). That is, characterize the computer based environment in which the users (designers, managers, etc.) will work.

An environment is the set of services provided to a user when a collection of tools are integrated together to form a cohesive set. We shall call the set of services provided by an integrated set of tools supporting a TSD Methodology a Total System Design Environment. Thus the TSD Environment forms a conceptual model of a group of services that support the first three stages in the life cycle of a project, with the support following a set of guidelines (from the appropriate methodology) to increase user productivity and system reliability. A high level characterization of the TSD Environment had to be developed in order to lay the foundation for the planning activities.

- Objective 2: Investigate design alternatives for the TSD Environment, and select a specific direction to elaborate. Apply the selected approach to develop a high level design proposal for a TSD System prototype.

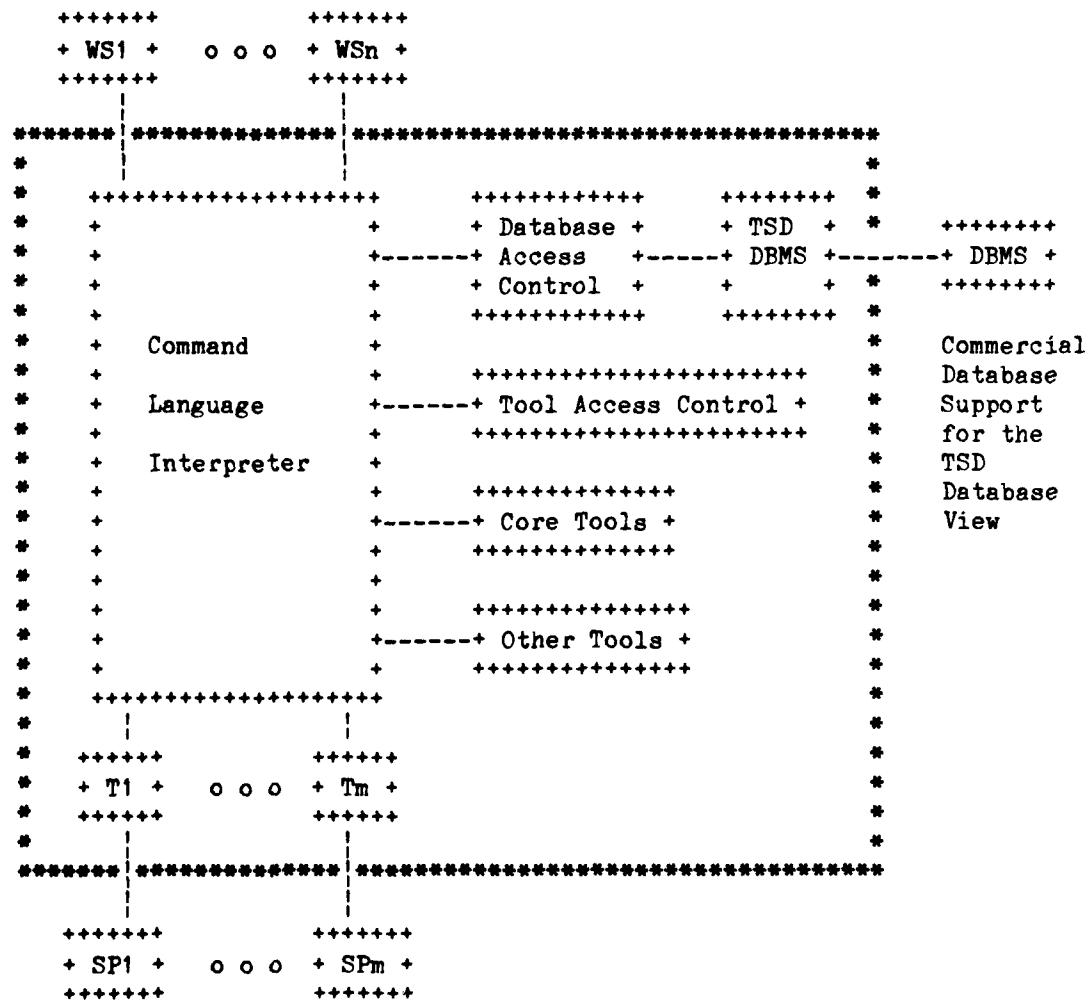
When a TSD System is installed in a particular computer center, unique features at that center may have to be accommodated within the TSD System itself. Special emulation facilities, unusual applications, or customized tools may all represent factors that may cause the basic TSD System to vary from one installation to another. These variations, however, represent local enhancements of the System, not basic changes. The collection of all of these possible enhanced versions of the System will be called the TSD System Family. A set of core tools, however, remains common among all TSD Systems.

- Objective 3: Develop a phased implementation plan for a TSD System prototype. Recommend ways to establish a TSD Facility supporting the prototype TSD System.

The prototype TSD System is recommended to be implemented as an outgrowth of the existing SAEF and by using currently available technology in order to obtain a running system within a reasonable budget of time and effort. The implementation plan is phased so as to allow the immediate exercising of parts of the overall system as they become usable. This approach increases the short term utility of the effort, and at the same time provides for critically important user feedback.

The TSD System prototype, when actually implemented, will also serve as an excellent test vehicle for the research and development necessary to create new tools and methodologies. Thus the implementation plan stresses feedback from both research and production efforts.

Work Stations Offering Specialized Design and Management Environments



Highly Specialized Local Resources (e.g., emulation engine, VLSI design tools, etc.)

TSD SYSTEM -- HIGH LEVEL STRUCTURE

ANNOTATED BIBLIOGRAPHY (Appendix A).

Abstracts for a large number of TSD-related government documents have been collected for use on future TSD projects.

GLOSSARY OF TERMS (Appendix B).

The glossary contains the definition of the key terms needed in order to understand the results of this study.

TSD GUIDEBOOK (Appendix C).

The guidebook provides in user oriented terminology a synopsis of the entire report, including descriptions of the TSD philosophy, existing and anticipated TSD technology, and the benefits derivable from its use in actual production.

ON REDUCING AMBIGUITIES IN METHODOLOGY DEFINITIONS (Appendix D).

Specification languages have an important role to play in the generation of unambiguous methodology definitions which, in turn, would affect the way in which configuration control and project planning would be carried out in the computer-aided design systems of the future. Precise methodology definitions hold the promise for better communication among designers and are also the key to increasing the designer's capacity to study, understand, evaluate, and compare one methodology against another. Furthermore, the inclusion of the methodology specification as part of the database of a computer-aided design system opens the possibility for a better enforcement of the correct use of methodology on a given project. Its use as an input to the project management tools is also being contemplated.

These opportunities are only now beginning to be explored and, to the best of our knowledge, no similar efforts have yet been reported in the open literature. The TSD assessment led to a proposal for a methodology definition language illustrated in Appendix D on a variant of the top-down program design methodology. Its use on the project has yielded significant quality improvements in the communication between the members of the research team. Many problems that were overlooked in the informal presentations of new proposals for the distributed systems design strategy have been rapidly uncovered during the effort of formally describing the methodology.

The language has the ability to describe the structure of and the relations between various products of the design process (configuration items), to capture changes in the state of these configuration items, to define consistency constraints between their states, and to prescribe the sequencing of design activities permitted by a methodology.

A FORMAL TREATMENT OF DISTRIBUTED SYSTEMS DESIGN (Appendix E).

Formal models have been developed for each of the specifications that are required by the system design stage in order to acquire a better understanding of the principles behind the TSD Methodologies. Formal models are provided for the system requirements constructed in the problem definition stage, for the processing model generated in the system architecture phase, and for the hardware/software requirements produced by the system binding phase. Concepts important in the TSD Methodologies (e.g., refinement, support, implementation, and binding) are also defined formally.

The development of these models represents an important contribution toward placing distributed system design on a solid formal foundation. As such, the models provide valuable guidance for the designer involved in the development and evaluation of certain classes of specification languages. They identify what concerns need to be addressed by the respective specification languages but not how they are addressed.

RIGOROUS APPROACH TO BUILDING SYSTEM REQUIREMENTS (Appendix F).

Because the ability to carry out the system design rests to a certain extent on the availability of a well-defined set of system requirements, the assessment included an investigation of formal methods for the specification of system requirements. The use of formal requirements is anticipated to play an increasingly important role in the TSD technology of the future. Our study looked into the feasibility of introducing the use of formal requirements definitions on system development projects.

A systematic approach to developing formal requirements by starting with the general model and by adapting it to the needs of the problem at hand has been proposed and illustrated by means of a simple but realistic example. The approach reflects the authors' experience with developing formal requirements for a variety of small scale problems. The notation used is based on set theory and predicate calculus both of which are generally considered essential in the education of the today's computer scientist and are familiar to many system designers. The conclusion is reached that, based on the experience accumulated with the use of both formal and semi-formal specifications, the development of formal requirements for small to medium size systems is feasible and can be cost effective.

FUNCTIONAL SPECIFICATION OF DISTRIBUTED SYSTEMS (Appendix G).

The potential for a major qualitative improvement in the effectiveness of systems development rests to a large extent on the availability of appropriate specification languages. While establishing the basis for precise communication, formal specifications also open the doors to extensive systematic (mental or automated) system design analysis techniques whose scope would ultimately include logical verification, performance checking, automatic generation of predictive models, and more. Such advances in system design technology are presumed to pave the way for the powerful development tools required by the TSD Facility.

A by-product of studying the technological needs of TSD Methodologies is the development of a formal Distributed Systems Design Language (DSDL). In DSDL, systems are described as nets of communicating processes. Each process in the net has its own local data over which it has sole control, has procedures that specify primitive and indivisible operations over the data, and possesses the ability to exchange messages with other processes in the net. The behavior of the process specifies the order in which its procedures are invoked. Sequences of procedure invocations, also called event sequences, are allowed to execute concurrently within the process.

A net is defined by its processes, by the logical communication links, and by the communication protocols associated with the individual links. Among the processes of a net, some are used to model its environment; they are called external processes. The links identify the logical connections between processes. Several processes may be associated with the same link and the same process may use several links. The way in which an individual link behaves is stipulated by the communication protocol associated with the respective link.

Several considerations have influenced heavily the nature of the DSDL: the emphasis on formality, the desire to promote the principle of separation of concerns, the need to support hierarchical specifications, and the aim toward generality. Formality is achieved through the use of set theoretical models for data representation, by employing predicate calculus in defining the procedures (using input/output assertions), etc. The principle of separation of concerns is reflected by the manner in which the definitions of the net and of the process are structured; they are meant to enhance the designer's ability to describe the system in terms of clean abstractions. Hierarchical descriptions of the system are enabled by the fact that processes may be refined into nets. Finally, the generality of the language is enhanced, among others, by its capacity to describe a variety of communication structures and protocols.

MODERN PROGRAMMING ENVIRONMENT ASSESSMENT (Appendix H).

The plans for a Modern Programming Environment (MPE) at DMA are assessed from the perspective of the TSD Facility. The rationale for this assessment lies in the fact that the MPE can be viewed as a TSD Facility specialized to the production of software at DMA. Since the plans for the far-term (Phase II/IIA) MPE development are likely to be affected greatly by the results of the near-term (Phase I/IA) development, this assessment is restricted primarily to the near-term plans. The objectives of the

assessment are discussed below.

- Objective 1: to evaluate current MPE plans and, if needed, prepare alternatives.

The current plans for the far-term MPE are considered sound for the most part. A reorganization of the tasks associated with the near-term development is proposed, however, in order to minimize the overall risk of this development. The proposed tasks for the near-term are: (1) the Facility Development task, which is concerned with the design, implementation, and phasing in of the near term experimental and full scale MPE facilities; (2) the Methodology Development task, which is concerned with the development and phasing in of a software development methodology for the MPE, and the preparation of requirements for methodology related enhancements to the far-term MPE; and (3) R&D Preparation for Phase II Startup, which is concerned with carrying out research and development in areas which are beyond the scope of the other two tasks but which are required for the startup of the far-term development effort.

- Objective 2: to identify issues which should be considered in future MPE efforts.

Many issues are identified which should be addressed in each of the three near-term tasks identified above. Issues relating to the Facility Development task include the maturing of the MPE tool set, the further development of user interface aspects, and the phased introduction of the MPE facility into the DMA production environment. Issues relating to the Methodology Development task include the identification of new tools needed to more completely support the life cycle activities identified in the methodology, and the phased introduction of the methodology. Issues which need to be addressed in the R&D task include the determination of evolving DMA software development needs, the determination of the effect of technology advances on the MPE, the development of structures and procedures to facilitate the evolution of the MPE, the specification of better management support tools, the determination of the appropriateness and feasibility of the multiple environment and project database concepts in the MPE, and the determination of the feasibility of achieving portability and vendor independence in the MPE.

2. TSD FRAMEWORK CONSOLIDATION

2.1 INTRODUCTION

The objective of this section is to report on the effort to consolidate the experience and knowledge accumulated as a result of past studies and projects that exercised the TSD concept and related technology with respect to their feasibility and the potential benefits they could bring to system design. The motivation for undertaking this task stems from the need to consider technological changes that have occurred since the TSD concept was first proposed, and with the expectation that important qualitative developments are at hand. The former requires a reexamination of the TSD concept in view of the continuing trend toward distributed processing and increased use of custom-made VLSI components, while the latter is based on preliminary results of several earlier studies.

The reexamination is aimed at ensuring that the TSD concept maintains its compatibility with the technological directions of this decade by giving proper recognition to any new scientific results pertinent to the TSD concept.

The qualitative improvements targeted for this study involve several important facets of the TSD concept, ranging from the very pragmatic to the highly abstract. They are a direct outgrowth of successful research and development activities that were carried out under the TSD umbrella. At a practical level, the emphasis is on expanding the ability to characterize in a precise manner a large class of TSD methodologies with respect to the entire system life-cycle, on enabling evaluation and comparison among different methodologies, and on providing the basis for a systematic approach to methodology development. In the realm of the abstract, special attention is given to reaching a better and more complete understanding of the fundamental decision-making processes involved in system design, particularly when H/S trade-offs are involved.

The starting point of the consolidation is the review of past TSD work and current state-of-the-art in system design. References to pertinent papers appear throughout this section, and an Annotated Bibliography of TSD-relevant government reports is available in Appendix A. The basis for the new unified and refined perspective on TSD is the notion of a methodological framework. It represents the means by which the consolidated TSD concept is formally defined. The TSD Framework is conceived as a methodological framework that synthesizes the fundamental attributes of the TSD methodologies in light of the philosophy behind the TSD concept. In turn, the TSD Framework is employed as an aid to sharpen the current understanding of the H/S trade-offs issue.

The results of the consolidation process are reviewed in the remainder of Section 2, which may be read as if it were a self-contained document. Its overall organization is as follows.

Section 2.2 introduces the TSD Framework and discusses its distinguishing features and basic philosophy. The framework is shown to be composed of several stages which represent groupings of phases. The phases, which are defined as activities taking place in some common knowledge domain and aimed at describing a transformation between two requirements specifications, are shown to possess a common structure, i.e., component steps. An approach for developing TSD Methodologies from the framework is also presented.

Section 2.3 considers the stages of the TSD Framework, one by one. The definition of each stage is given in terms of its component phases. Each phase is defined with respect to the requirements specifications it uses and produces. Each pertinent step is analyzed and the nature and complexity of the techniques required to support it are identified. Whenever such techniques are available, the reader is advised. Each phase description concludes with a discussion of the nature of the specifications generated by the respective phase.

Section 2.4 is an elaboration, in the context of the framework, on the topic of H/S trade-offs. Emphasis is placed on explaining the dynamic nature of H/S trade-offs, an approach which is in strong contrast with the earlier static perception of the way the H/S trade-offs are carried out.

Section 2.5 gives precise definitions for four key technical aspects of the system life-cycle (development, analysis, enhancement, and maintenance), and identifies the connection between them and the TSD Framework in preparation for the assessment being carried out in Section 3.

Section 2.6 contains a summary of conclusions.

2.2 TSD FRAMEWORK DEFINITION

INTRODUCTION

The software crisis of the 70's played a significant role in increasing the general awareness of, and interest in, design methodologies. In particular, it brought about a wide-spread belief that large system development without strong methodological support involves unacceptable risks. As a result and in a relatively short time span, major advances have been registered in the areas of methodology development, project control and review, specification techniques, and automated documentation and analysis tools [CHAN78, WASS78, WEGN79, YEH77].

Nevertheless, serious problems continue to plague the software industry. Some problems are due to a reluctance of management and personnel to accept change, a reluctance that is partially justified by the high cost of retraining and retooling. Some problems are due to the failure of many highly touted techniques to deliver all that was advertised. Finally, there are problems due to the fact that the level of abstraction being dealt with in the areas of methodology development, analysis, and evaluation is too low. This manifests itself through the existence of few generally accepted principles, through parochialism, and through a limited ability to evaluate and compare proposed methods.

Matters have been further complicated by an ever increasing interdependency between hardware and software. This has led to the view that a system is a hardware/software (H/S) aggregate in which the hardware and software aspects must be treated together and not separately as has been traditional. Today's system designer must have an understanding of both hardware and software and must have an appreciation of their combined impact on the performance characteristics of the total system. In particular, the designer needs a unified methodological perspective in which hardware and software issues can be brought together properly.

This report presents an approach for satisfying two pressing methodological needs, namely, the need for a more abstract treatment of methodologies and the need for a unified methodological perspective for hardware and software. The report introduces a type of model called a methodological framework for handling the first need, and proposes a particular framework, called the Total System Design (TSD) Framework, for handling the second need.

A methodological framework is an abstraction of a class of system design methodologies. The framework is hierarchical in structure, being composed of stages which are, in turn, composed of phases, which are composed of steps. The stages represent broad design areas such as system design, software design, and hardware design, while the phases represent finer divisions of these design areas. For example, a stage dealing with software design could contain separate phases for software architecture, program design, and coding. The steps represent design activities that go on within the design areas. They include activities such as performance evaluation, functional verification, documentation, and acceptance. The

framework can describe, in a straightforward manner, the logical organization and the design activities intrinsic to a particular methodology. This fact makes the framework a potentially valuable analytic tool for comparing the fundamental traits of different methodologies. It also makes the framework useful as a specification tool for describing a methodology that is to be designed.

The TSD Framework is intended as a specification for system design methodologies which have a unified perspective of hardware and software and which embody other attributes necessary for effective and efficient design. Briefly stated, these methodologies (1) recognize formally the H/S dualism, (2) avoid premature hardware selection, (3) minimize error costs through early error detection, (4) treat performance constraints as a major driving force behind the design process, (5) promote design automation, and (6) assure proper attention to human interfaces.

The remainder of this section is devoted to the TSD Framework. The purpose is two-fold: to introduce the structure of the TSD Framework, and to illustrate thereby the nature of methodological frameworks in general. The exposition is introductory in nature, with a detail description of the TSD stages, phases, and steps being given elsewhere (Section 2.3).

The discussion is organized as follows. The next subsection concentrates on the description of stages and phases. While most of them are quite mundane in concept, the system design stage contains some novel aspects. They are the result of an emphasis placed on avoiding premature hardware and software selection. Another subsection is dedicated to the steps recognized by the framework. It is shown there that all phases involve the same ten steps. While some have been recognized for a long time (even if not presented from the same perspective), others represent a departure from traditional views. The inference step, for instance, formalizes the process of evaluating the design decision taken in one phase with respect to technological implications on subsequent phases. Originally motivated by the H/S partitioning issue, inference has been shown to be present in all phases. Another example is the treatment of integration as a step rather than a phase. The integration activities are distributed among phases based upon the nature of the expertise required to carry them out.

The presentation continues with a discussion of the relation between the structure of the framework and its six stated objectives. It is also pointed out that distinct methodologies may emphasize different objectives and thus instantiate the steps in different ways. This particular aspect is further clarified in a subsection which illustrates the use of the framework for purposes of methodology development. By selecting a sample application area and by considering its characteristics and their relation to both technology and application environment, a methodology is derived in a systematic manner from the TSD Framework. The approach suggests that, for each application area and organization, methodology development involves a certain degree of "pre-design" in addition to the selection of particular techniques for design, analysis and specification. Conclusions and references appear at the end.

STAGES AND PHASES

Figure 2-1 shows the logical structure of the TSD Framework. The stage boundaries are drawn along traditional lines and the concern of each is obvious from the stage name. Each stage is composed of two or more phases which represent well known design areas. The downward arrows represent requirements specifications that define the problem to be solved by a subsequent stage. Each specification has two parts, a functional requirement and a set of implementation constraints. The upward arrows indicate the flow of finished products during the integration portion of system development. The idea here is that each stage is responsible for the integration of its portion of the design. The integration process thus begins at the lowest level of detail and works upward until all components of the system have been assembled and tested. Although the diagram does not show it, the reader should visualize upward and downward arrows between the phases of a stage. They have been omitted from the diagram in order to make it more readable.

The dependency between phases is not as simple as the figure might suggest. For example, it should not be inferred that a project must complete each phase or stage before beginning the next phase or stage. Parts of a project may move through the development process faster than other parts and hence be in different phases and stages. Also, methodologies represented by this framework can differ in the way they schedule the basic activities of the framework and in the design techniques that they employ. These distinctions must be kept in mind at all times in order not to read into the framework more than it represents.

The PROBLEM DEFINITION STAGE is composed of two phases, called identification and conceptualization. Both phases are application domain dependent and their successful completion rests on a good understanding of the application. The IDENTIFICATION phase is informal in nature and has an exploratory flavor. Its objective is to produce an identification report which contains all the information available with regard to the system support required by the application at hand, as well as any relevant constraints. Despite the fact that the level of formalization and abstraction of the identification report is relatively low, the report serves two important functions: it establishes the communication link between the designer and the user and provides the necessary base for the development of a formal definition of the problem. This formal development is done in the conceptualization phase.

The CONCEPTUALIZATION phase uses the identification report in order to generate the system requirements. These requirements contain a conceptual model which formalizes the system's role from a user perspective and the application constraints identified earlier. Because of its formal nature, the conceptual model provides a solid basis for the entire design process and represents the ultimate correctness criterion against which the final system is judged. The ability to meet all the stated constraints is a second fundamental evaluation criterion.

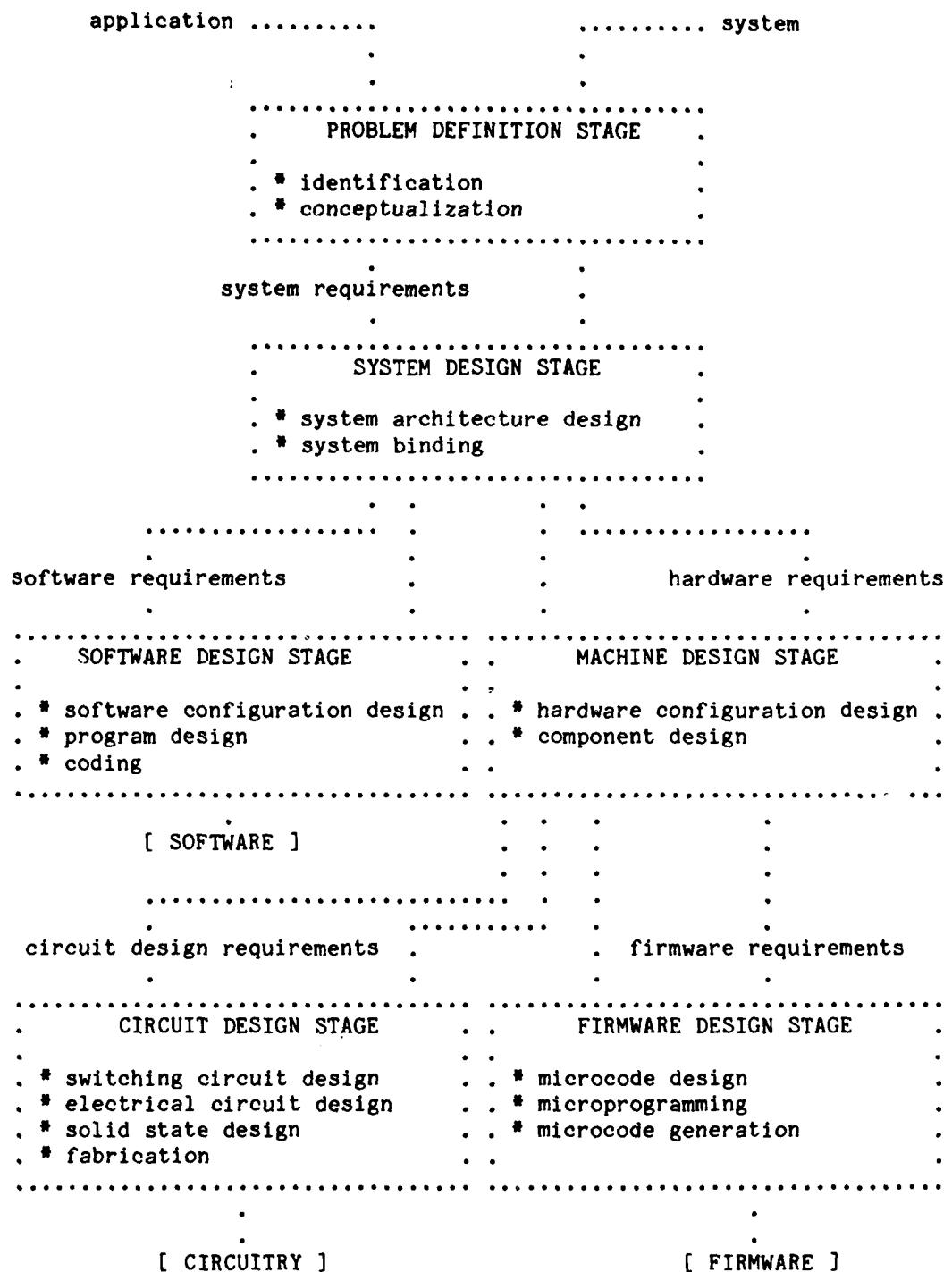


FIGURE 2-1: TSD FRAMEWORK STRUCTURE

The SYSTEM DESIGN STAGE is central to the TSD Framework because H/S trade-offs are one of its major responsibilities. System architecture design and system binding are the two phases that make up this critical stage. The main concern of the SYSTEM ARCHITECTURE DESIGN phase is to investigate system design alternatives and their potential impact on the choices for a feasible system configuration (i.e., H/S mix). Without making any explicit choices with respect to the selection of particular software or hardware components, this phase is involved in the performance of H/S trade-offs to the extent that design decisions taken here affect the class of feasible configurations in a manner too significant to be left to chance.

The functional/performance specifications generated by the system architecture design, as part of the system configuration requirements, form the basis on which a particular H/S mix is selected during the SYSTEM BINDING phase. The hardware and software requirements being generated by this phase may assume a variety of H/S combinations from off-the-shelf complete systems to custom built components. The election of one option over another is determined by the nature of the system design, the constraints it has to meet, and the available technology. It is accomplished by the system architecture design phase, but the selection of specific components is done during binding.

The SOFTWARE DESIGN STAGE includes all activities relating to software design and procurement. There are three phases involved in this stage. The first one, SOFTWARE CONFIGURATION DESIGN, is responsible for the procurement of off-the-shelf software as well as the overall high level design of the software system. The software requirements are the basis for these activities which result in the development of program requirements specifications, including the complete design of its data and environment interfaces. The PROGRAM DESIGN phase, in turn, takes these requirements and produces the program design (data and processing structures) which, together with all pertinent assumptions and constraints, make up the implementation requirements. They are used by the CODING phase to build the actual programs.

The MACHINE DESIGN STAGE plays a role similar to that of the first two phases of the software design stage. The HARDWARE CONFIGURATION DESIGN phase is concerned with the procurement of off-the-shelf machines and the design of the high level architecture of custom hardware. Component requirements are developed for all entities that are part of the custom hardware and passed on to the COMPONENT DESIGN phase. This phase generates a register transfer level machine description that will be included in the circuit design requirements and in the firmware requirements.

The CIRCUIT DESIGN STAGE follows a generally accepted scenario involving four phases: SWITCHING CIRCUIT DESIGN, ELECTRICAL CIRCUIT DESIGN, SOLID STATE DESIGN, and FABRICATION. Each phase generates design requirements for the phase listed after it.

The FIRMWARE DESIGN STAGE consists of three phases that are an analog to program design, coding, and compilation. These phases are called MICROCODE DESIGN, MICROPROGRAMMING and MICROCODE GENERATION.

STEPS

The previous subsection gave a general introduction to the design areas covered by the TSD stages and phases. This subsection gives a general introduction to the activities that occur during the design process. The major design activities within a phase are called STEPS. There are ten steps which collectively represent the activities within any phase, regardless of the nature of the phase. Some of the steps represent activities that are common practice among good designers and appear to be fundamental to the design process. The other steps represent activities that are needed to meet the objectives of the TSD Framework. The names of these steps are listed below. The dashed lines are used to indicate groups of related steps.

formalism selection
formalism validation

exploration
elaboration
consistency checking
verification
evaluation
inference

invocation

integration

The FORMALISM SELECTION step encompasses the activities involved in selecting a formalism for a particular problem domain. Candidate formalisms are evaluated for their expressive power in that domain and also for qualities such as simplicity of use, lack of ambiguity, analyzability, and potential for automation. While this step must take place before other steps in the phase, it often occurs long before them. This is sometimes due to the use of a methodology that is based on a particular formalism, but is more often simply a matter of policy or is due to the availability of tools tailored to that formalism.

The FORMALISM VALIDATION step encompasses activities involved in determining whether a formalism has the expressive power needed for a particular task. It also includes the evaluation of formalisms from the standpoint of ease of use. These tasks are generally non-trivial and may involve both theoretical and experimental evaluations. Theoretical results may indicate the power and the fundamental limitations of the formalism while past experience with it on similar projects may provide insight in its appropriateness and ease of use. The step also includes evaluations of the formalism's potential for design automation (as a way to bring about productivity increases) and its ability to support hierarchical specifications (as an aid to controlling complexity).

The EXPLORATION step encompasses the mental activities involved in synthesizing a design. These activities are creative in nature and depend on experience and natural talent. They cannot be formalized or automated unless the problem domain is restricted to a significant degree.

The ELABORATION step encompasses the activities involved in giving form to the ideas produced in the exploration step. In general, this step involves the use of formalisms and its activities are facilitated by design aids such as text editors and formatters. This step includes the building of a concrete object such as a piece of hardware.

The CONSISTENCY CHECKING step encompasses activities such as checking for incorrect uses of formalisms, checking for contradictions, conflicts, and incompleteness in specifications, and checking for errors of a semantic nature. It includes checking for consistency between different levels of abstraction in a hierarchical specification and the reconciliation of multiple viewpoints.

The VERIFICATION step encompasses activities involved in demonstrating that a design has the functional properties called for in its requirements specification. Since each phase has a requirements specification and produces a design, this step applies to all phases. A common example of this type of activity is the proving of program correctness. The difficulty of this task is well known and is also representative of the difficulty of the verification task in general.

The EVALUATION step encompasses activities involved in determining if a design meets a given set of constraints. This includes constraints which are part of the requirements specification for the phase and constraints which result from design decisions. The nature of the evaluation activities depends on the type of constraints being analyzed. They include classical system performance evaluation of response time and workload by means of analytical or simulation methods; deductive reasoning for investigating certain qualitative aspects like fault tolerance or survivability; construction of predictive models for properties such as cost and reliability.

The INFERENCE step encompasses activities involved in assessing the potential impact of design decisions made in the phase. The domain of these activities include: impact on the application environment, ability of subsequent phases to live with decisions made in this phase, effect on system maintainability and enhanceability, effect on implementation options. While these issues must be considered in every phase, proper treatment is particularly critical in those stages defining architectures.

The INVOCATION step encompasses the activities associated with releasing the results of the phase. It includes quality control activities where tangible products are involved and review activities leading to the formal release of output specifications. It is this latter aspect that gives the step its name, since the release of specifications in effect invokes subsequent phases.

The INTEGRATION step encompasses the activities associated with the configuring and testing of that portion of the total system that was designed in the phase. Although it is traditional to consider integration to be a design area that would qualify as a stage in the framework, the integration activities have been distributed among the phases in recognition of the fact that the expertise needed to test that portion of the system is the same as the expertise needed to design it. In addition, design errors found during integration must naturally be referred back to that phase. It is therefore fitting that integration be considered a phase activity.

THE TSD FRAMEWORK OBJECTIVES REVISITED

There are three factors that have influenced the conception of the TSD Framework: current state-of-the-art in the area of distributed systems design, the conviction that a systematic application of the principle of separation of concerns is fundamental to a formal treatment of design methodologies, and a set of six explicit methodological objectives motivated by generally desirable features of good system design and by the desire to make the selection of hardware and software on the basis of more objective criteria than those in use today. The six methodological objectives are: (1) formal recognition of the H/S dualism, (2) deterrence of premature hardware selection, (3) minimization of error costs through early error detection, (4) proper consideration of the role played by performance constraints in the design process, (5) design automation, and (6) proper attention to human interfaces.

The TSD Framework builds directly on the current understanding of system design methodologies with respect to both the phases and the steps that make up its structure. Its steps represent a taxonomy of the design activities generally encountered in system design. Its phases, aside from those included in the system design stage, have been recognized already by other authors. There are, however, two important distinctions between the way phases and steps are used here and elsewhere. First, the grouping of activities into a phase is based upon the nature of the technical expertise they require rather than upon considerations related to project management. The latter are relegated to methodologies and are not part of the framework. Second, the steps are abstractions over classes of design activities and not specific actions to be carried out by the designer in some prescribed order. These differences stem from the fundamental distinction between frameworks and methodologies.

The criteria used in the selection of both phases and steps are a direct reflection of the principle of separation of concerns. The traditional separation between hardware and software design, for instance, is captured by the identification of distinct phases associated with each. At the same time, however, because judicious partitioning of the system functions between hardware and software demands the two to be considered together and to perform certain trade-offs, the system design stage has been included. It separates the selection and specification of the hardware and software from hardware and software design.

Because the TSD Framework has been used primarily as a way of specifying a class of distributed system design methodologies called the TSD Methodologies, some of the characteristics required of these methodologies have affected the structure of the framework. The manner in which this took place is explained below.

Formal recognition of the H/S dualism and the desire to avoid premature selection of the hardware led to the proposal of the system design stage in which design decisions take into consideration the fact that a given system function may be realized in hardware, software, or by a combination of the two. These two related objectives also represent the original motivation behind the introduction of the inference step which, in the context of the system design stage, evaluates the consequences of

system level design decisions with respect to the hardware/software selection options they may promote or rule out. Furthermore, the hardware/software dualism suggested the adoption of similar structures for the software and machine design stages.

The minimization of error costs is supported, at the framework level, by the emphasis on formal specifications which are the foundation for computer-aided design systems able to carry out cost-effective and, at the same time, extensive automated error checking. The presence of the verification and consistency checks define the nature of the error detection to be incorporated in the TSD Methodologies.

The role played by performance constraints is made explicit in the structure of the requirements generated by the various phases and in the definition of the evaluation step. Moreover, the definition of the integration step includes checking the satisfiability of the constraints and the validity of the performance models employed in the evaluation step. (The checking that takes place in the evaluation step is only as good as the models used and the accuracy of the assumptions made about the performance characteristics of components to be designed in subsequent phases.)

The definition of the formalism selection and validation steps have been strongly influenced by the intent to support the TSD Methodologies by means of computer-aided design. The promotion of formal specifications is, to the largest extent, due to the emphasis on design automation.

Finally, in order to stress the importance of properly evaluating the design of the human interfaces, the inference step includes an investigation of the potential impact of alternate design decisions upon the user environment and the evaluation step includes human engineering studies.

FROM FRAMEWORK TO METHODOLOGY

Because effective methodologies are application and environment (i.e., organization) dependent, the TSD Framework specifies the requirements for not one single methodology, but a class of similar yet distinct methodologies. Differences between methodologies that address the needs of different applications and organizations manifest themselves in the relative weights attached to the importance of the methodological objectives of various steps, in the order in which the design activities are scheduled, in the frequency and extent of the design checks, etc. Consequently, an in-depth understanding of the nature of the systems to be developed and of the character of the system development and maintenance organizations is a prerequisite to considering an instantiation of the framework.

The framework may be used as a methodology skeleton and checklist which is pruned and refined during methodology development based on the nature of the application and organization. A certain amount of "pre-design" takes place. It may *a priori* remove from consideration some technological alternatives, it may restrict the designer to using particular specification languages thus eliminating the formalism selection and validation steps, etc. To illustrate this process and as an aid to exposition, the following application is used as an example. We will assume that the systems to be developed are turnkey systems for relatively small data processing applications. The development and maintenance of these systems is to be the responsibility of the vendor organization, and copies of the same system are to exist in several user organizations.

Methodology development begins by identifying those stages and phases that are unnecessary. For the given application, the circuit design and firmware design stages are eliminated by virtue of the fact they deal with high cost design and maintenance components, both in terms of needed personnel expertise and required facilities. The machine design stage is also discarded because, in order to keep maintenance costs down, it is desirable to limit the type of hardware to a single machine. The nature of the application, small data processing, makes it possible for the vendor organization to select a single minicomputer as the common hardware support for all systems to be developed. Since hardware selection is done *a priori*, the machine design stage is totally unnecessary.

Next, the remaining stages and phases are analysed with respect to the role they might have to play in the new methodology in light of the specific application being considered. The investigation reveals that the objective of the system architecture design phase is limited to the determination of how to allocate the system's functions among one or more minicomputers of a given type. The binding phase, in turn, is assigned the task of evaluating the proposed distribution against the characteristics of the actual machines and of generating the hardware and software requirements. The former specify the number of machines and the way in which they are configured. The latter contains a description of the software to be placed on each of the machines, the implementation language (always the same), and the communication protocols between the software pieces residing on different machines.

Most activities abstracted by a given phase depend heavily on the nature of the formalism chosen in the formalism selection step. Because the use of identical formalisms on all system design projects has obvious advantages, several specification languages could be adopted as company standard after evaluating their appropriateness for the type of projects being envisioned. The formalism selection and validation steps are thus eliminated from the methodology. In the context of our example, the vendor organization could decide in favor of: some graphic language [ROSS77, ROMA79] to assist the conceptualization phase with the functional decomposition of the system requirements; a data processing system design language [TEIC77] for both the system architecture and software configuration design phases; some form of pseudocode for the program design phase; and a standard programming language for coding.

Following the formalism selection above, the steps that are involved in the evaluation of the specifications produced by each phase need to be defined in terms of the intended scope, objective, and analytical techniques to be used. It may turn out for instance that consistency checking and verification steps are carried out by means of some nonautomated procedures [ROMA79]; the evaluation step in the conceptualization phase is implemented as a user review; the evaluation step in the architecture design phase is limited to questions of time and space and done by hand, while in the software design stage it is neglected completely; and the inference step is not present anywhere due to lack of adequate techniques and tools.

Besides the use of particular techniques, another factor that contributes to the effectiveness of some methodology is the manner in which design activities identified by the framework as steps within various phases are to be sequenced on actual projects. Considering the example again, project control objectives may dictate that all relevant phases are to be done in the order in which they appear in the framework except for the case when corrections to earlier work are deemed necessary. Different subsystems, however, are permitted to be in different stages of development as long as their interfaces are clearly identified. On the other hand, within a single phase, all steps are to be repeated, in the same sequence as in the framework, for every level of the hierarchical specification being produced. (Significantly more complex sequencing strategies have been observed in some existing design methodologies [MCCL75].)

Methodology development must also include the managerial perspective on system design, which brings into the methodology aspects not yet considered. They would deal, at a minimum, with issues related to project status evaluation checkpoints and procedures (e.g., reporting and auditing procedures), system design and integration planning, physical and human resources allocation, marketing strategy, etc.

Finally, once a methodology has been developed, there is still the problem of acquiring a facility which, through its tools and personnel, enforces the methodology, speeds up tedious and time consuming human activities, assists in project control, etc. In other words, the facility complements effective methodology and management with a highly productive design environment brought about by the availability of automated tools.

CONCLUSIONS

By concentrating solely on the methodological schema identified by the framework, one is more apt to see the real goals, strengths, and weaknesses of a methodology. Thus, empirical comparisons among methodologies may be complemented in the future by evaluations on an abstract level. (Unfortunately, the use of the framework as an analytic tool has been investigated, so far, only to a very limited extent.) Furthermore, a change in goals may be better effected by first subjecting the framework to needed enhancements or refinements and only later making the corresponding adjustments to the methodology itself. Modifications to the methodology need to consider the way in which the changes in its foundation (i.e., framework) relate to the characteristics of the application, the techniques supporting the methodology, and the environment in which the methodology is used. This approach to methodology development and enhancement promises to be less prone to judgmental errors, and promises to provide valuable assistance to designers investigating methodological alternatives. The approach has been successfully employed already in the development of a class of distributed system design methodologies.

REFERENCES

- [CHAN78] Chandy, K. M. and Yeh, R. T. (editors), Current Trends in Programming Methodology, vol. 3, "Software Modeling," Prentice Hall, 1978.
- [MCCL75] McClean, R. K. and Press, B., "The Flexible Analysis Simulation and Test Facility: Diagnostic Emulation," Technical Report TRW-SS-75-03, TRW, Redondo Beach, CA. 90278, 1975.
- [ROMA79] Roman, G.-C., "Verification Procedures Supporting Software Systems Development," 1979 NCC Proc., pp. 947-956, June 1979.
- [ROSS77] Ross, D. T. and Schoman, K. E., "Structured Analysis for Requirements Definition." IEEE Trans. on Soft. Eng. SE-3, No. 1, pp. 6-15, January 1977.
- [TEIC77] Teichroew, D. and Hershey, III, E. A., "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems." IEEE Trans. on Soft. Eng. SE-3, No. 1, pp. 41-48, January 1977.
- [WASS78] Wasserman, A. I. and Belady, L. A., "Software Engineering: The Turning Point," Computer, pp. 30-41, September 1978.
- [WEGN79] Wegner, P. (editor), Research Directions in Software Technology, MIT Press, 1979.
- [YEH77] Yeh, R. T. (editor), Current Trends in Programming Methodology, vol. 2, "Program Validation," Prentice Hall, 1977.

2.3 STAGES IN THE TSD FRAMEWORK

2.3.1 PROBLEM DEFINITION STAGE

TERMINOLOGY OF THIS STAGE

REQUIREMENTS: User Problem Statement
PHASE: Identification
REQUIREMENTS: Identification Report
PHASE: Conceptualization
REQUIREMENTS: System Requirements

DESCRIPTION OF STAGE ACTIVITIES

One of the most critical aspects in developing a new system is to insure that the problem the system is designed to solve is the same one that the user of the system wants solved. This implies that the customer (assumed to be skilled in the area of the application) must accurately communicate his needs to an analyst (assumed to be skilled in the area of computer systems analysis). Since these two specialists speak different technical languages, how can the communication gap be bridged? Further, how can the correctness of the resulting information transfer be assessed? The first stage in the TSD framework insures that this communication takes place and that the result is an accurate, mutually acceptable definition of the problem to be solved. This objective is achieved in two phases: first by a general identification of the problem's characteristics, and then by the articulation of these characteristics as a more formal conceptual model.

At the end of this stage, a System Requirements report is produced that presents the total set of functional specifications and performance constraints for the creation and evaluation of the ultimate delivered product. Thus, the overall success or failure of a project hinges on the successful completion of the Problem Definition Stage, as effected through its Identification and Conceptualization phases.

STATE-OF-THE-ART

Successful achievement of this stage's goals requires that the proposed member methodologies have certain attributes. Among these is the ability to support a formal approach to the problem definition activity, with the resulting definition free of any design bias; moreover, the methods must clearly separate constraints on the system from its functional requirements. Additional tools required to support the use of these methodologies include utilities for text input, editing and formatting, database storage and retrieval of text, formal syntax verifiers, report generators (including system consistency checkers and verification aids), and possible functional simulators for system verification and evaluation.

The usual practice today is to treat the Identification Report as a basis for immediately starting design activities, thus completely avoiding the establishment of a formal conceptual model of the proposed system. Design specification languages are used for expressing whatever system conceptualization still occurs, resulting in the introduction of a premature design bias into the developing system. This is particularly evident when a high-level design language is used to propose a solution at this stage.

It is clear that considerable effort still is needed to develop appropriate formalisms and tools to support conceptual model building, since without them consistency checking and model verification remain error prone. As errors are allowed to pass from one stage to the next, they require more and more effort to correct. This fact alone could justify the TSD framework requirement that any methodology used in this first stage must produce a formal conceptual model that completely defines all system requirements.

PHASE NAME: Identification

PURPOSE

Identify and define the requirements and constraints needed to specify completely a computer system that will solve a customer's problem.

INPUT

A general problem statement plus a list of customer personnel available for contact.

OUTPUT

Identification Report detailing the problem requirements and constraints as obtained from the customer.

STEPS

FORMALISM SELECTION

The Identification Phase represents the first contact point between the customer and the builders of the proposed system. This first phase must create an atmosphere conducive to the free exchange of information between all personnel concerned, since the final report of the phase must consider all factors pertinent to the problem definition. During the entire system life cycle, each phase will require a specific language or formalism to express the pertinent information, and certainly this phase is no exception. However, this beginning phase is unique due to the breadth of both potential problem requirements and personnel experiences. Hence, it may be best to handle it in a more informal manner. This implies that English text should be the vehicle used to express the relevant assumptions, constraints, and demands to be satisfied by the proposed system. Note that there is no design done in this phase; all effort is concentrated on the identification and specification of the problem requirements.

Although a natural language does not present the opportunities for rigorous analysis associated with a more formally defined language, some structuring should still be imposed on the English text. Forms, checklists, and suggested report outlines have all been proposed as mechanisms that may help overcome the ambiguity of English text and thus help insure that all of the necessary factors are considered [NAUM80]. This point will be discussed further in the later steps, and also is considered in [HENI79, ROMA79, TAGG77].

FORMALISM VALIDATION

The technical vocabulary should be drawn primarily from the application area, since the language of the customer should best define the problem during this phase. Computer jargon is not the basis for effective communication between the customer and the analyst.

EXPLORATION

Just what are the factors that need to be established during the Identification Phase? A number of references [METZ73, STAA79] present suggested checklists that help to fulfill the abovementioned needs. These lists represent attempts to generalize experiences based on the results of past design efforts. As such they must be considered as guidelines adaptable to the actual application at hand. Specific results will emerge as the customer and the analyst work together to explore the broad possibilities and requirements of the general problem space being considered. Detailed interviews with customer personnel represent the main source of information for this step. Questions that need to be addressed include:

- Why should a system be built?
- What assumptions are being made to define the system?
- What customer needs must be satisfied?
- What constraints must be imposed on the system?
- What environmental demands must the system satisfy?
- Which system boundaries are hard or soft?
- What trade-offs at what costs may be allowed?

One result of this initial exploration step is the establishment of tentative system boundaries and the corresponding definition of human/system interfaces.

ELABORATION

As the exploratory activities conclude, and the overall scope of the system has been identified, the points thus raised must be elaborated by filling in sufficient detail to define completely all necessary system functions and constraints. This requires the drafting of a report that integrates all information collected thus far. This activity may be facilitated by appropriate tools for text entry, editing, and formatting.

The information recorded in this step represents an important part of the final Identification Report. Hence considerable care must be given to insure that it is complete, accurate, and unambiguous. However, it must also represent the beginning of a project database that will support all project-related activities for the duration of the entire system life cycle. The information entered into the project database supports an audit function and serves as a source of authority for all later system developments. That is, any factor in the later stages must be able to trace its reason for existence back to entries

created in the project database during this phase and, ultimately, to the customer's original problem statement.

CONSISTENCY CHECKING

Despite the lack of formalism during this phase of the TSD framework, it is still of vital importance to attempt to check the system elaboration for consistency. For example, the requirements established by one system function should be compatible with the requirements of another function or of any of the imposed constraints. Further, the elaboration should be complete, particularly in the sense that there should be no undefined terms, functions, or constraints. The sooner such errors of validity/consistency are detected, the easier (i.e. cheaper) it is to correct them.

VERIFICATION

Once it is decided that the requirements are consistent, the requirements should be verified to any extent possible. The key element in this verification is for the customer to agree that the desired problem has been completely and accurately identified in terms of a usable set of requirements. That is, an Identification Report has been produced that contains a complete and unambiguous identification of the problem. Proposed user scenarios represent a possible means of testing for any problems. If the system identification is not satisfactory, then more time and effort must be devoted to this phase by both the customer and the analyst. Although a more rigorous system verification will be possible in the next phase, at this point there are some additional tools that may be available to aid in the review and acceptance of the requirements document. These include feasibility studies, simulated scenarios, and comparisons based on extrapolations from existing systems.

EVALUATION

The next step in this phase must be an evaluation of the work done so far. It is still early in the TSD framework, and thus any results tend to be soft. However, it still may be possible to build cost and other forecasting models to support the analytical task of the next step.

INFERENCE

The impact of the identified system requirements and constraints on later stages of the TSD framework are considered here. There is no detailed system design available yet, but still the background and knowledge of the project team should allow some assessment of the technical feasibility of achieving the desired goals. Further, the studies conducted for the verification and evaluation steps should allow additional conclusions to be established relative to the

technical and economic consequences of the system requirements. Thus, at this point, estimates should be established for schedules, staffing, and resource requirements for all of the later system development stages. In addition, this information should establish the effect of the final system on the user, in terms of cost, time, environmental impact, and resource requirements. The results of the cost/benefit analysis enables the customer to decide on final acceptance of the Identification Report.

INVOCATION

The next phase will be invoked when the Identification Report, consisting of all system requirements and constraints, is completely accepted by all concerned. The invocation step consists of passing the requirements document to the Conceptualization Phase for the development of a more formal system model.

INTEGRATION

The final step of any phase is the acceptance and integration of the results obtained from the invocation of the later phases. Since the Identification Phase is the first phase in the TSD framework, its final result consists of putting into production the complete system as specified by the Identification Report. Thus, system installation at the user's site, user testing, and user training must all be accomplished. Maintenance procedures and methods for phasing the new system into active production must also be specified and implemented.

PHASE NAME: Conceptualization

PURPOSE

Convert an informal set of requirements for the proposed system into a formal conceptual model.

INPUT

Identification Report containing a complete description of the system requirements and constraints.

OUTPUT

System Requirements containing a formal conceptual model of the proposed system plus the set of all constraints that the system must satisfy. The requirements must be complete and unambiguous since it must first serve as a basis for all later development work and then as a yardstick for testing the acceptability of the final delivered system.

STEPS

FORMALISM SELECTION

Once all of the aspects of the informal system identification produced in the previous phase have been accepted, it is time to develop a more formal conceptual model of the proposed system. This will require the selection of an appropriate formalism, along with a validation that the particular formalism can handle problems from the given application area. Many suggested formalisms have been described in the literature, and many of these are still undergoing active development. (For some of the most widely used current systems see: [GANE79, ORR77, ROSS77, TEIC77].) The published systems vary widely in expressive power, ease of use, extent of automation, and tool availability [LISK79]. Hence, the formalism selected for this phase will have a strong impact on the future progress of the project. Even if a special purpose system must be designed and implemented, it is important for the TSD framework that the informal requirements from the previous phase be converted into formal specifications that can drive later stages.

Using a formal approach to building a conceptual model may be justified by noting a number of advantages. A formalism implies that a definite syntax has been established, thus allowing automated tools to do syntax checking, reporting, and project database maintenance. The selected formalism must have the ability to model all aspects of the application domain, and so it certainly depends on the state of the art in that area. However, an application specific formalism may include semantics such that additional verification checks, particular to the

application, also may be automated. Examples of this approach may be seen in the database field, where extensive data models have been developed. The relational data models are especially advanced from the point of view of including semantic information and constraints, so that the formal models even provide for automating much of the design process itself [ULLM80].

Note that this stage in the TSD framework is attempting to define completely and unambiguously the system requirements. The vehicle for this definition is a conceptual model of the proposed system that includes all necessary functions and constraints. This model, however, does not represent a system design! A major drawback in many of the available formalisms is that they were created initially as program specification languages. This heritage forces the user of these systems to consider (perhaps unconsciously) design aspects of the evolving conceptual model. In this early phase there is simply not enough information to make any proper design decisions.

FORMALISM VALIDATION

Most of the published techniques have additional significant weaknesses when considered from the TSD framework point of view: They tend to have evolved from a data processing background with a bias towards generality. The resulting formalisms are not application oriented and thus are difficult to apply to a specific problem. They frequently use visual (flow-chart like) presentations of essential system relationships that drastically reduce any possible automation and make formal consistency checking and verification techniques very awkward to use. Further, a background based on large data processing systems means that most of the formalisms are very weak in the areas of real time signal processing and hardware/software trade-off considerations. As a consequence, they are weakest in exactly the features needed most for embedded computer systems.

EXPLORATION

The exploration and elaboration steps of this phase parallel the informal efforts of the previous phase, except that the formalism now allows much more precision in the definitions of information flows, functionalities and system constraints. Although the potential system user does not need the expertise to develop descriptions in the selected formalism, it is important for him to be able to read and understand such descriptions. The customer must agree that the conceptual model being created does indeed meet the application needs (as were stated in the informal description created in the previous Identification Phase), since the Report produced in this Phase will be the technical driving force behind all subsequent system design efforts. Further, the trace of what formal requirements were induced by what informal statements must be maintained through the project database for later potential authorization, feedback and modification.

ELABORATION

The results of the previous exploration step must be recorded in an appropriate format. Construction of a conceptual model on a digital computer, using the selected formalism, represents that format. However, it is important to maintain the viewpoint that the model is being built to help in understanding the evolving system design. That is, there should be a definite bias towards clarity and ease of understanding in all aspects of building the model.

It is in this step that the full power of the formal approach begins to be used. The same types of tools mentioned in the previous phase are of use here, but since they are being applied to a formally defined system the possibilities for automatic error detection are much greater. In addition, this step forms the foundation required to automate many parts of the next three steps.

CONSISTENCY CHECKING

One major benefit of casting the conceptual model into formal terms comes from the availability of automated tools that help to insure the internal consistency of the model. In particular, this phase can benefit greatly by tools that process input data for proper syntax and/or semantics and reports on the status of various information flows, functional dependencies, and constraint specifications. What information is created and never used? Multiply created? Used but never created? Constraints never referenced? Are all interfaces compatible? Is there consistency among levels in a hierarchical model? Many such questions may readily be answered given an appropriate supporting formalism. Further, the ability to change the conceptual model and immediately see the overall effect by using these automated reporting tools allows the analyst to do a far better and faster job of creating an acceptable system model.

VERIFICATION

The functional requirements built into the conceptual model must match the requirements specified in the Identification Report. Checking that every external aspect is covered by the model implies that informal statements and formal statements must be established as equivalent - this is a necessary task, but one that the current state of the art cannot handle automatically. At least, functional simulation and user reviews provide a solid foundation of information for the verification of the conceptual model.

EVALUATION

The evaluation and inference steps also benefit by the introduction of the selected formalism. These steps in the Conceptualization Phase become much more quantitative than in the previous Identification Phase, thus increasing the overall confidence

level in being able to produce the desired system. As in the previous phase, the evaluation step should look at the raw data from feasibility studies, simulations, and system application scenarios (usually obtained during the last two steps) to determine the behavior of the proposed system. Analysis of the resulting system characteristics produces the system evaluation for this development stage.

INFERENCE

The information on system behavior analyzed in the Evaluation Step also must be studied to determine how the system requirements will impact the later TSD framework stages. The System Requirements Report must contain a complete, unambiguous, testable set of requirements and constraints that the customer agrees will establish the formal basis for all later system development. If an earlier step finds a requirement or constraint that cannot be satisfied, or the inference step suggests a later stage may not be implementable, then the feedback within this stage still has a chance to correct the situation. Additional studies on cost effectiveness, scheduling, etc (all started in this step of the previous stage) may now be expanded based on the more quantitative information developed from the conceptual model.

INVOCATION

Once the system requirements have been specified by the acceptance of the formal conceptual model, the System Design Stage of the TSD Framework is invoked.

INTEGRATION

The Integration Step consists of testing the deliverable system, as created by the next Stage, to determine if all of the specifications and constraints detailed in the System Requirements Report have been satisfied. Thus this step concludes with a complete system demonstration.

REFERENCES

- [GANE79] Gane, C. and Sharson, T. Structured Systems Analysis: Tools and Techniques, Prentice-Hall, 1979.
- [HENI79] Heninger, K. L. "Specifying Software Requirements for Complex Systems: New Techniques and Their Applications," Proc. Conf. on Specifications of Reliable Software, April 1979.
- [LISK79] Liskov, B. H. and Berzins, V. "An Appraisal of Program Specifications," Research Directions in Software Technology, MIT Press, pp. 276-301, 1979.
- [METZ73] Metzger, P. W. Managing A Programming Project, Prentice-Hall, 1973.
- [NAUM80] Naumann, J. D., Davis, G. B. and McKeen, J. D. "Determining Information Requirements: A Contingency Method for Selection of a Requirements Assurance Strategy," J. Systems and Software 1, pp. 273-281, 1980.
- [ORR77] Orr, K. T. Structured Systems Development, Yourdon Press, 1977.
- [ROMA79] Roman, G. "Total System Development Framework," Dept. of Computer Science Report WUCS-79-10, Washington Univ., St. Louis, 1979.
- [ROSS77] Ross, D. T. "Structured Analysis (SA): A Language for Communicating Ideas," IEEE Trans. Software Engineering SE-3, January 1977.
- [STAAT79] von Staa, A. and Cowan, D. D. "The Development Proposal: The First Step in Software System Construction," IBM Research Report RZ969 7/27/79.
- [TAGG79] Taggart, W. M. and Tharp, M. O. "A Survey of Information Requirements Analysis Techniques," Comp. Surveys 9, No. 4, December 1977.
- [TEIC77] Teichroew, D. and Hershey, E. A. "PSL/PSA: A Computer Aided Technique for Structured Documentation and Analysis of Information Processing Systems," IEEE Trans. Software Engineering SE-3, January 1977.
- [ULLM80] Ullman, J. D. Principles of Database Systems, Computer Science Press, 1980.

2.3.2 SYSTEM DESIGN STAGE

TERMINOLOGY OF THIS STAGE

REQUIREMENTS: System Requirements
PHASE: System Architecture Design
REQUIREMENTS: Binding Requirements
PHASE: System Binding
REQUIREMENTS: Software Requirements, Hardware Requirements

DESCRIPTION OF STAGE ACTIVITIES

The system design stage defines the logical structure of the system and the manner in which hardware and software are to be used in its implementation. These activities are split between the following phases:

SYSTEM ARCHITECTURE --- defines the structure of the system and all system-level processes.

SYSTEM BINDING --- selects the hardware that is to support the system processes.

The primary input to this stage is the system requirements specification generated in the Problem Definition Stage. This includes: a conceptual model of the role of the system in the application environment, performance requirements such as throughput and response time goals, physical constraints such as limitations on size and power consumption, reliability requirements such as survivability goals, and design guidelines such as restrictions on types of equipment, technology, and vendors.

In addition to the system requirements, the design process is guided by recognized rules-of-the-trade and by good engineering practice. These include design guidelines which promote the development of systems that are easy to maintain and enhance. Architectural models that reduce the impact of hardware obsolescence on system life-cycle are emphasized.

Customer interactions comprise a third type of input to this stage. These interactions occur for many reasons, including: clarification of ambiguous or incomplete specifications, revision of conflicting or unachievable requirements, and assessment of the impact of a proposed design on the user environment. The latter may require the construction of mock-ups and the development of simulated versions of the system. It may also require new studies of the application environment. These activities are an intrinsic aspect of the system design stage.

The design activities in this stage are disciplined in a manner that meets the common objectives of all TSD methodologies. Briefly stated, these are

- Systematic approach to hardware/software trade-offs.
- Systematic approach to system/environment interfacing.

- Systematic approach to early detection of errors.
- Systematic approach to performance evaluation.
- Emphasis on maintainability and enhanceability.
- Emphasis on computer aided design.

The output of the System Design Stage has many parts, including: a model identifying all system-level processes, including interfaces and performance constraints; a complete description of all system-level algorithms, including goals, underlying assumptions, and proofs of correctness; a description of the physical structure of the system, including a description of all processing hardware, physical links, and communication protocols; a complete mapping of system-level processes to system hardware. In short, it includes all information needed for the design or procurement of hardware and software (the Hardware Requirements and the Software Requirements) and all information needed for system integration, maintenance, and enhancement.

STATE-OF-THE-ART

The activities of this stage require certain resources, formalisms, and software tools in order to be carried out efficiently. With regard to resources, the design activities should be supported by a computer system which provides database support for the storage of the design data, and designers should have interactive access to this system from terminals with graphics capabilities. With regard to formalisms, there should be standard formalisms for each aspect of design documentation. Standardized formalisms are necessary to the development of unambiguous documentation and are also fundamental to the development of software tools. With regard to the latter, a variety is needed to expedite the documentation and analysis activities of the design process. These include

- utilities for checking syntax
- utilities for checking consistency
- utilities to perform or assist in verification
- documentation aids such as a text editor/formatter
- utilities for generating performance data from processing models

These resources are commercially available from a wide range of vendors and in a multitude of configurations. Most companies involved in system design have these resources in one form or another, and current technology is capable of outfitting almost any design environment that might be defined.

Many of the necessary formalisms are available due to the considerable attention that system design specification languages have received during the last decade. Proposals range in flavor from standardized graphic representations, tables, and document formats [ROSS77] at one extreme, to formal languages having well-defined syntax and semantics [ROBI77] at the other.

The work on program specifications dominates the field in terms of attention received and level of formality. A good survey of available formal program specification techniques is given in [LISK79]. Those specification languages that are designed to support concurrency, such as Path-Pascal [CAMP79] and DREAM [RIDD78], are appropriate for the specification of distributed systems.

Also available but somewhat less formal are RSL [BELL77] and PSL/PSA [TEIC77]. These are particularly noteworthy because they have been implemented and are being used in the development of large systems. Both are part of computer-aided design packages which provide database support for storing design specifications and provide software tools tailored to the specification formalism. The services provided include consistency checking, automatic generation of system simulations, reporting, configuration management, etc.

The specification of distributed systems continues to have many unresolved problems. Some of these are due to an incomplete understanding of what to include in a functional specification. Others are due to an incomplete understanding of how to relate issues such as concurrency coordination and input/output specifications which, despite their interdependence, are currently being treated in an independent manner. Another source of problems is in the area of performance specifications. Except for the work of Booth and Wiecek [BOOT80], there has been little research in this area.

PHASE NAME: System Architecture Design

PURPOSE

The purpose of this phase is to define the architecture of the system. It assumes a general model in which the architecture is represented as a set of communicating processes that reside on a network of processors. The term "processor" as used here is a general term representing all entities that can support processes. It includes graphics terminals, CPUs, special purpose hardware, complete computer systems, etc. The architecture phase defines a processor structure and a set of processes that will meet the goals contained in the system requirements. This includes a specification of function and performance requirements for each process, and a specification of device type and implementation constraints for each processor and for each interconnection. In most cases, the hardware specification does not uniquely identify all details of the hardware, but describes only those aspects needed to support the functionality and performance of the system. It is the task of the Binding Phase to determine the exact hardware that is to be used.

INPUT

The input data for this phase is the system requirements specification produced by the Problem Definition Stage. It consists of a conceptual model, which defines the role of the system in the application environment, and a set of associated design constraints. This data includes (but is not limited to) the following items.

Conceptual Model

- A description of the services to be provided by the system.
- A description of the system interface.

Constraints

- Performance requirements such as throughput rates and response times.
- Physical constraints such as limitations on size, weight, and power consumption.
- Reliability requirements such as survivability goals.
- Equipment constraints such as restrictions on types of equipment, technology, and vendors.

OUTPUT

The output of this phase is the system-level information needed for binding, integration, maintenance, and enhancement. This information includes the following items.

Binding Requirements

- A processing model defining system-level processes and their interactions, interfaces, communication protocols, and performance constraints. Model includes response of processes to undefined data and/or protocol violations wherever such is possible.
- Implementation constraints consisting of an assignment of processes to processors and a specification of device type, technology, and selection constraints for the processors and interconnections.

Theory of Operation

- A description of all system-level algorithms, including goals, underlying assumptions, and proofs of correctness.
- All models used to predict performance characteristics of the design.
- A description of the design decisions reflected in the architecture, including motivations, underlying assumptions, and interdependencies.

STEPS

FORMALISM SELECTION

A formalism with the following characteristics is needed for the representation of processing model information:

- It should be easy to understand and use. This reduces the risk of a design specification describing more or less than intended.
- It should be able to represent the types of processing structure common to the application area. In some cases this may only require an ability to represent finite state machines; in others, it may require an ability to represent networks of cooperating processors.

- It should make possible the expression of functional and performance information in the same model. This eliminates the risk of inconsistencies between models.
- It should be syntactically and semantically well-defined. This reduces the risk of ambiguous specifications and makes possible the detection of certain types of specification error.

FORMALISM VALIDATION

Those aspects of the selection criteria that are mathematical in nature may be validated by formal analysis. Those that are subjective or dependent on the application domain must be judged on the basis of experience in the application area.

EXPLORATION

The development of an architecture requires that algorithms be devised for performing the system functions and that a processing model be devised for executing these algorithms. This development effort is guided by the need to meet the requirements and constraints given in the system requirements specification.

The design process is also guided by general rules-of-the-trade which suggest structures that facilitate maintainability and enhanceability. Also considered are implementation issues. This is because the processing model and associated parameters will determine the set of implementation choices, and the design must therefore be guided toward a reasonable set of options.

The development of the architecture normally proceeds in an incremental manner, with certain aspects taking shape before others. In order to assure that the developing architecture is consistent with the design goals, TSD methodologies require that each increment be formally documented and validated before being incorporated in the architecture. This acceptance process consists of the activities described in the steps called Elaboration, Consistency Checking, Verification, Evaluation, and Inference.

ELABORATION

The task of this step is to create a formal representation of the design. Besides the obvious need for appropriate formalisms, there is a need for documentation aids such as those listed below.

- Interactive terminals with graphics capabilities.
- A database system for storing designs.

-- Software for detecting typographical errors. An example is a program that lists names that occur only once, these being potential typos.

CONSISTENCY CHECKING

This step involves checks of various sorts. At the simplest level, there are checks for syntactic correctness of specifications and checks for agreement of interface specifications of communicating processes. At a more complex level, there are checks to verify that a refinement is consistent in function and performance with the item being refined, and checks for agreement of complementary specifications such as a data-flow model with a behavioral model.

While the simplest of these checks can be readily carried out by software tools given the current state-of-the-art, this is not the case for the more complex. They are best dealt with by semi-automatic approaches in which the designer performs the checks with the aid of software tools.

VERIFICATION

The task of this step is to show that the design is consistent with the intent of the system requirements specification. Mechanization of this task is beyond the current state-of-the-art and verification must therefore be carried out by informal means, sometimes with the assistance of the user. This may be a permanent situation since the items of information being compared tend to belong to different levels of abstraction.

One very important means of verification is through trace-driven simulations and through simulations in which the user interacts with the system. An example where both are warranted is a flight-training system for pilots. Such simulations can also be used to evaluate the appropriateness of proposed system/environment interfaces.

EVALUATION

This step determines the extent to which the design meets the constraints imposed on it. This applies to all categories of constraint, whether given in the system requirements specification or identified as the consequence of design decisions. The needed analysis can sometimes be performed by analytic means, but most often requires the use of functional and/or discrete event simulation (or emulation). Some simulations may require user interaction and some may have to be trace-driven or distribution-driven.

In general, it is desirable for simulations to be derived directly from the processing model by software tools [BELL77, TEIC77]. This speeds up the evaluation process and also eliminates a major source of error by taking humans out of the loop. Automatic derivation of

simulations seems feasible for the more common types of analysis but seems unlikely for the rest. In cases where new models must be created in order to perform a particular analysis, the models and all underlying assumptions must be recorded in the design documentation of the system.

INFERENCE

The purpose of this step is to make sure that the design has properly addressed the issues of implementation options, maintainability and enhanceability, and environmental impact. This involves the following assessments:

- The structure is evaluated for modularity, simplicity of interconnections, and low performance requirements. Anything that would appear to unduly restrict the range of implementation options, either by limiting the choice of technology or by requiring non-standard or specialized components, is rejected.
- The processing model is evaluated from the standpoint of modularity, degree of process interactions, and standardization of interfaces. Anything that would unduly complicate either maintenance or enhancement is rejected.
- The overall design is reviewed from the standpoint of impact on the user environment. Any interactions between system and environment whose details were not dictated by the system requirements specification are referred to the user for approval.

INVOCATION

This step consists of a formal review and sign-off on the architecture, thereby giving official permission to start the binding phase.

INTEGRATION

This step has two tasks. The first is to configure a complete prototype system and make it operational. This validates the quality and completeness of the documentation and sets the stage for the second task, which is to verify that the prototype has the function and performance required by the system requirements specification.

PHASE NAME: System Binding

PURPOSE

The purpose of this phase is to produce hardware and software specifications for the system. The architecture phase has already done much of this work, and this phase simply finishes the task. With regard to software, most of the software requirements are provided in the processing model produced by the architecture phase. All that remains is to specify the target machine, and this can be done as soon as the hardware portion of this phase has been completed. With regard to hardware, the task is more complex.

The architecture phase views the system as being supported by a network of processors, where the term "processor" is a general term that includes all forms of hardware, including computer terminals, CPUs, special purpose hardware, complete computer systems, etc. That phase also establishes the basic nature of the processors and the interconnections. For example, one processor might be described as a mini-computer with certain characteristics, another processor might be described as a custom device to be implemented in CMOS technology, and an interconnection might be described as a serial link with a certain bandwidth. These specifications sometimes uniquely define the component, but more often they simply identify a class of components. The task of the binding phase is to select a specific implementation when more than one option exists.

This selection process is guided by a variety of considerations, many of which are system-wide in scope. As a result, selection cannot be done by considering each processor and interconnection in isolation. Instead, candidates must be identified for each processor and interconnection and then selections must be made that are best for the system as a whole. The following items are representative of the factors that are taken into account.

- maintenance. This biases the selection toward minimizing the number of vendors.
- purchase cost. This is the sum of hardware and software costs for the entire system. It takes into account the fact that higher costs for some items may be offset by lower costs for others.
- operating cost. This takes into account power consumption, cooling costs, and costs of service contracts for the entire system.
- availability and manufacturer's reputation. This takes into account delivery times, product quality, and ability of the manufacturer to assist when problems occur.

INPUT

The input data for this phase is the binding requirements specification produced by the System Architecture Phase. It consists of a processing model, which defines the system-level processes, and a set of implementation constraints. The nature of this information is indicated below.

Processing Model

-- A model defining system-level processes and their interactions, interfaces, communication protocols, and performance constraints. Model includes response of processes to undefined data and/or protocol violations wherever such are possible.

Implementation Constraints

-- An assignment of processes to processors and a specification of device type, technology, and selection constraints for processors and interconnections.

OUTPUT

The output of this phase consists of two parts, a hardware requirements specification and a software requirements specification.

The hardware requirements specification contains all technical information needed for procuring existing (off-the-shelf or build-on-demand) hardware and for letting contracts for the design and development of custom hardware. The existing-hardware category includes existing computer systems and customizable hardware.

Hardware that must be designed from scratch is specified in terms of its functionality, performance, and implementation constraints. This includes a complete logical and electro-mechanical description of its interfaces. In those cases where the custom hardware is to support software, the hardware/software interface will be defined well enough to allow the concurrent and independent design of the software.

The software requirements specification contains the technical information needed for the procurement of existing software packages and for the letting of contracts for the design and development of custom software. Software is specified in terms of its functionality, performance, and implementation constraints. The implementation constraints contain a description of the hardware/software interface, with the term "hardware" being understood to include computer systems. It may also specify that the software be written in a particular high-order-language or assembly language.

STEPS

FORMALISM SELECTION

In the hardware area, specification formalisms are needed for describing hardware that is to be designed. For programmable devices, the formalisms must deal with the instruction-set domain, while for non-programmable devices, the formalisms must deal with functional domain. In the software area, specification formalisms are needed for describing software that is to be designed.

Several specification formalisms exist for each of the areas described above. Each formalism is well suited to a particular range of application, and all application areas appear to be covered.

FORMALISM VALIDATION

The suitability of a formalism depends on how well it meets the needs of the application area. In most cases, there is no standard describing these needs, so suitability must be judged on the basis of experience.

EXPLORATION

The identification of suitable candidates for a given processor can be a complex task. Each processor has been assigned one or more system processes by the architecture phase, and each process has a set of performance constraints that must be met. In the case of custom hardware, the assigned processes define the behavior of the hardware to be designed. In the case of programmable hardware, the assigned processes define the software that is to be purchased or designed and they also constrain the selection of the programmable hardware. The latter results from the fact that software performance is dependent on the instruction set and speed of the hardware. In those cases where the software already exists, performance can be ascertained experimentally. In those cases where the software must be written, the suitability of the hardware can be judged by two approaches. The first is by educated guesses based on experience with similar processes and hardware. The second is by identifying the performance-critical sections of the processes and then programming them for the hardware under consideration.

The selection of winning candidates can also be a complex task. This is particularly true when the item in question is a computer system because the candidates usually have features not called for in the design effort. Since these features are potentially useful, it is unreasonable to simply disregard them. At the same time, it is difficult to know how to weigh them in the selection process. This issue has received much study [TIMM73] and is still unresolved.

Binding can be done in a brute-force manner by exhaustively identifying all candidates for each processor and interconnection and then performing the selection process. This is very time-consuming, especially for large systems, and more efficient approaches are needed. One possibility is to use the system-wide considerations during candidate identification to reduce the number of candidates that must be considered in detail. Such a strategy would have to be worked out very carefully in order to assure that the only candidates that would be eliminated are those that would be eliminated by the brute-force approach.

ELABORATION

The task of this step is to record the design data generated during the binding phase. These data include the models used for evaluating candidates, the considerations used in choosing winning candidates, and the specifications generated for hardware and software. This documentation task is facilitated by documentation aids of the type described under the elaboration step of the system architecture phase.

CONSISTENCY CHECKING

This step deals with checking the software and hardware requirements for internal consistency and proper use of the respective formalisms. Most effort is expended in the evaluation of the interface consistency between software components, between hardware components, and between software and hardware.

VERIFICATION

The verification of the software and hardware requirements is carried out against the binding requirements received from the system architecture design phase.

EVALUATION

The task of this step is to determine whether potential implementation methods can meet the performance requirements of the processes being bound. If the implementation being considered is an off-the-shelf package, the performance parameters of the package must be reconciled with the requirements of the process(es) it is to implement. If the implementation is custom software running on some device, the issue is whether software that meets the performance requirements of the process(es) can be written for that device. Techniques for determining this are discussed under the exploration step of this phase.

INFERENCE

The objectives of this step are similar to those of the corresponding step from the system architecture design phase. The difference is in the fact that here the actual machines to be used are known and, therefore, the analysis becomes more concrete.

INVOCATION

This step consists of a formal review and sign-off on the hardware and software specifications, thereby giving official permission to begin procurement of hardware and software.

INTEGRATION

This step is vacuous for the binding phase. All aspects of system integration fall under the purview of the system architecture phase.

REFERENCES

- [BELL77] Bell, T. E., Bixler, D. C., and Dyer, M. E., "An Extendable Approach to Computer-Aided Software Requirements Engineering," IEEE Trans. on Soft. Eng. SE-3, No. 1, pp. 49-60, January 1977.
- [BOOT80] Booth, T. L. and Wiecek, C. A., "Performance Abstract Data Types as a Tool in Software Performance Analysis and Design," IEEE Trans. on Soft. Eng. SE-6, No. 2, pp. 138-151, March 1980.
- [CAMP79] Campbell, R. H. and Kolstad, R. B., "Path Expressions in Pascal," Proc. 4th Intl. Conf. on Soft. Eng., pp. 212-219, 1979.
- [LISK79] Liskov, B. H. and Berzins, V., "An Appraisal of Program Specifications," Research Directions in Software Technology, P. Wegner (editor), pp. 276-301, MIT Press, 1979.
- [RIDD78] Riddle, W. E., Wiledon, J. C., Sayler, J. H., Segal, A. R., and Stavely, A. M., "Behavior Modeling During Software Design," IEEE Trans. on Soft. Eng. SE-4, No. 4, pp. 283-292, July 1978.
- [ROBI77] Robinson, L. and Levitt, K. N., "Proof Techniques for Hierarchically Structured Programs," CACM 20, No. 4, pp. 271-283, April 1977.
- [ROSS77] Ross, D. T. and Schoman, K. E., "Structured Analysis for Requirements Definition," IEEE Trans. on Soft. Eng. SE-3, No. 1, pp. 6-15, January 1977.
- [TEIC77] Teichroew, D. and Hershey, III, E. A., "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems," IEEE Trans. on Soft. Eng. SE-3, No. 1, pp. 41-48, January 1977.
- [TIMM73] Timmreck, E. M., "Computer Selection Methodology," Computing Surveys 5, No. 4, pp. 199-222, December 1973.

2.3.3 SOFTWARE DESIGN STAGE

TERMINOLOGY OF THIS STAGE

<u>REQUIREMENTS:</u>	Software Requirements
<u>PHASE:</u>	Software Configuration Design
<u>REQUIREMENTS:</u>	Program Design Requirements
<u>PHASE:</u>	Program Design
<u>REQUIREMENTS:</u>	Implementation Requirements
<u>PHASE:</u>	Coding

DESCRIPTION OF STAGE ACTIVITIES

The Software Design Stage transforms the software functionality and performance requirements, as specified by the System Design Stage, into a working software system meeting those requirements. This transformation may generally be performed in one of three ways: the custom design, development, and implementation of the software system, the procurement of a suitable software system from an external source, or some combination of these two activities, supported by a rigorous integration and validation effort.

STATE-OF-THE-ART

Recognition of the well-known software crisis came during the late Sixties. Since that time, considerable progress has been made toward systematizing the production of reliable software. Starting with tentative steps toward the improvement and definition of programming style [KERN74], the ideas of top-down design, modularity, structured programming, and stepwise refinement [WIRT71a, WIRT73, DAHL74] crystallized into well-defined practices supported by a variety of vehicles ranging from guidelines [JACK75] to formalized rules for describing and documenting programs (see [LISK79] for a general discussion of the state of the art), and automated tools to expedite and monitor the program implementation process [TEIC77]. The impact of this movement extends to the programming languages themselves, since their construction, stemming from earlier perceptions of the programming process, generally did not anticipate the trend toward systematization. As a result, important changes have been introduced into existing languages such as FORTRAN [ANSI66] and COBOL [ANSI68]. In addition, concern with structured programming has prompted the appearance of new languages such as PASCAL [WIRT71a, JENS75], Simula [DAHL66, DAHL70], Ada [DOD79], and CLU [LISK77a] in which such features as strong typing and user-defined abstract data types play prominent roles.

Response to the software crisis transcended the individual program, addressing a spectrum of system design issues as well. Accordingly, the system design and development process is currently supported by a variety of technical and management instruments, including chief programmer teams [BAKE72, BAKE73], structured analysis [ROSS77a, ROSS77b], and structured walkthroughs [YOUR75]. In summary, the software development process is

now perceived as an engineering endeavor, with the primary emphasis on reliability, maintainability, quality assurance, and fault tolerance.

However, further work is needed since many of the above-mentioned areas are still in their early stages of development. Issues yet to be addressed include:

- The further integration of automatic tools with methodologies well suited to their use, and integration between different methodologies and techniques.
- Greater transfer of technology between developers and potential users, as well as further use of existing tools and techniques.
- The further development of formal languages and rigorously verifiable techniques. These should have a positive impact on the traceability of requirements and constraints throughout the system development process.
- A better understanding of how to exploit concurrency in the design of software systems, all the while managing the complexity of such designs. Accordingly, the need for formal tools and sophisticated design aids oriented toward concurrent systems presents an important frontier for future research.

PHASE NAME: Software Configuration Design

PURPOSE

The purpose of the Software Configuration Design phase is the development of a model of the software system structure, description of the programs comprising the software system, the interfaces through which the components communicate with one another and with their environment, and a specification of how the components of the system are to be acquired (i.e., through custom design and implementation, off-the-shelf procurement, a combination of the two, or by some other means).

INPUT

The Software Configuration Design Phase receives as its input from the System Binding Phase of the System Design Stage a set of Software System Requirements. This consists of the system functionality specifications and the system constraints. The former describes what the system must do; the latter imposes quantitative and qualitative restrictions on the set of possible design solutions. Examples of such constraints are: performance constraints, hardware and configuration constraints, and implementation language constraints.

OUTPUT

The Software Configuration Design Phase provides the Program Design Phase with a specification of the structure of the software system, functional and performance specifications of the programs which need to be designed to implement that structure, and the program interfaces. The latter refer to major software system interfaces, such as databases and file management systems, major data structures, user interfaces, etc. This phase may also develop some further constraints to be observed in the later software development phases, such as time and space constraints for each system program.

STEPS

FORMALISM SELECTION

The formalism selected for this phase of the system design should be useful in the description of the important design aspects of a software system. These include the structure of the system, behavior of the component programs, the interfaces between the component programs, and procedural relationships and models.

The criteria to be employed in the selection of one formalism over another include: degree of formality, nonprocedurality, verifiability, analyzability, the existence of automated support tools, simplicity, the support of hierarchical descriptions, suitability for the description of concurrency in a system, and the capability to deal with

multi-level, virtual-machine system architectures.

Existing formalisms having some of these properties, in different and varying combinations, include:

- Structured Analysis (SA, or SADT) [ROSS77a, ROSS77b]
- Structure Charts and Data Flow Charts [MYER73]
- HIPO's (Hierarchy plus Input-Process-Output) [IBM74]
- Structured Systems Analysis [GAN79]
- Procedural and data abstractions [LISK77b]
- PSL/PSA [TEIC77]
- HOS [HAMI76]
- RSL [BELL77]

FORMALISM VALIDATION

The important factors to be considered in the validation of the selected formalism are: the formal aspects of the system selected and their applicability to the problem domain, and empirical evidence obtained through previous experiences with the use of the selected formalism. In addition, the formalism must be suitable for software engineers to use comfortably as a design tool.

EXPLORATION

Guidelines to be employed in the decomposition of the software system into a set of programs include:

- Isolation of functionally-related or data-related activities into single programs, thereby maintaining strong correspondence between conceptual activities and actual processes.
- Use of straightforward, well-documented interfaces to minimize apparent complexity.
- Employment of stepwise refinement to develop the design from initial requirements in a systematic manner.
- Comparison of different possible program and interface configurations to arrive at the best possible design.

Available techniques and aids include:

- Top-down design through stepwise refinement [WIRT71b, DAHL72].
- Bottom-up design through stepwise composition [DIJK68b].

- The use of virtual machines to implement a set of interacting software layers [DIJK68b].
- Modularization of data and function [PARN72b].
- Data structuring and encapsulation [LISK79].
- Invariant data structures.

ELABORATION

The current trend in elaboration centers around the use of one of the design aids mentioned above. Some of the design aids mentioned are supported by automatic tools; more generally, however, they are simply a set of rules and procedures to be followed. Common software aids such as databases and computer graphics, however, can be helpful in the management of information, particularly on very large projects. Application of these aids expedites the production of requirements definitions for the Program Design Phase.

CONSISTENCY CHECKING

The specifications produced in the elaboration step should be checked against the rules of the selected formalism (i.e. syntax). Problems with self-consistency, contradictions, and completeness should be checked for, as well as consistency between levels of the development when hierarchical specifications are used. Interface usages should be verified against their definitions. When complementary specifications are used, they should be checked against one another (e.g., behavior vs. structure; data flows vs. event sequences).

VERIFICATION

The logical correctness of the specifications should be verified with respect to the specifications input to the stage. This includes the verification that no information from the previous stage is lost or ignored. Logical verification is somewhat similar to a proof of the correctness of a program, but an order of magnitude more difficult.

EVALUATION

The evaluation step examines the specifications set down in the previous steps and produces data describing aspects of those specifications. This may include an evaluation and prediction of the performance of the selected software architecture, in terms of the performance of each component program. This evaluation may be arrived at through various methods of performance modeling and simulation, forecasting models, reliability models, etc. The specifications should also be examined in light of the other qualitative and quantitative

constraints, such as fault detection and recovery, maintainability, flexibility, transportability, freedom from problems such as deadlock, and feasibility of implementation. Other important factors are the human engineering factors, which may be studied by a user review of human interfaces, mockups, etc.

INFERENCE

In the inference step, those aspects from the mass of data produced in the evaluation step relating to the user environment and the next phases are studied and evaluated to obtain a global picture of the quality of the proposed software system architecture. The initial study of testing strategy is also begun. In addition, the impact of the proposed design on the module level design phase should be considered (e.g., the gross complexity of the algorithms to be used). The results of this inference determine whether the next design phase should be invoked, or whether further iteration over and refinement of the design are necessary. The potential impact of the parts of the system specified for procurement should also be anticipated at this point.

INVOCATION

The Program Design Phase is invoked.

INTEGRATION

The tested and 'debugged' programs are received back from the program design phase, and are integrated together, along with any off-the-shelf software specified in the software system architecture, into a coherent software system. The system is thoroughly tested, using test cases generated in previous steps of this phase, along with any testing suggested by designers in the lower phases. Testing can be speeded and aided in completeness by use of any of the several automatic testing aids and systematic testing procedures available [HETZ73].

PHASE NAME: Program Design

PURPOSE

The Program Design Phase is responsible for the decomposition and refinement of the program specifications generated in the previous phase into lower-level, reasonably sized, completely specified modules, including descriptions of the algorithms and the local data structures to be employed.

INPUT

The inputs to the Program Design Phase are simply the outputs of the Software Configuration Design Phase, the Program Design Requirements, consisting of:

- the system structure
- the program requirements specifications
- the program interfaces.

OUTPUT

As output, the Program Design Phase produces a set of Implementation Requirements. These requirements are:

- the module specifications, composed of algorithm specifications and local data structure specifications
- the intermodule interfaces, such as data structures and parameter lists.

STEPS

FORMALISM SELECTION

The most important criteria for formalism selection in this phase are the degree of formality of the formalism, and the ease with which it can be used. Some examples of formalisms relevant to this phase of the design are: English, conventional flowcharts, structured flowcharts [NASS73], schematic logic [JACK75, JENS79], pseudocode (with assertions), decision tables, finite state machines, and formal program specification languages (e.g. PSL [TEIC77], GYPSY [AMBL77]) and techniques (axiomatic specifications [HOAR69], operational specifications [PAGA81]).

FORMALISM VALIDATION

The formalism should be examined primarily with regard to the degree to which specifications produced through its use may be verified, and the productivity expected from its users. Productivity is largely a function of how well the formalism is suited to use by software engineers, and how easily it can be translated to code.

EXPLORATION

The purpose of this step is the decomposition of the program into a set of functionally simple modules, and the selection and description of the data structures and algorithms to be employed in the implementation of the program modules. Techniques useful in this decomposition are: modularization [PARN75]; structured programming [DAHL72]; stepwise refinement [WIRT71b]; abstract machines [DIJK68b]; data structuring; abstract data types and encapsulation [LISK74]; and information hiding [PARN72b].

ELABORATION

In this phase, the elaboration step simply involves the description of the design decisions made during the exploration step through use of the selected formalism(s).

CONSISTENCY CHECKING

Consistency checking is performed with respect to the rules of the formalism, i.e., the syntax of the formalism. The use of the interfaces also has to be checked, both between modules, and when the modules must interface to the environment. The module descriptions must be checked for violation of invariants during their processing, and while interacting with the program interfaces.

VERIFICATION

The specification must be verified against the requirements of the programs as passed in from the software design phase. They can be checked for the preservation of specified I/O assertions. Correctness proof techniques may be employed, to the extent that they are usable and useful.

EVALUATION

The evaluation step of this phase is concerned with many of the same issues as described in the evaluation step of the software configuration design phase. The further decomposition should be reflected through the further refinement of the performance parameters.

INFERENCE

In the inference step, one of the things to be considered is the implication of the data structures selected on the choice of a programming language. Different structures are more naturally represented and manipulated in different languages. Likewise, the relation between the proposed algorithms and the operations they require, and the features and performance of different programming languages should be considered.

INVOCATION

The Coding Phase is invoked.

INTEGRATION

The program design phase receives back tested modules from the Coding Phase. Integration involves the testing of these modules as they interact with each other as programs.

PHASE NAME: Coding

PURPOSE

The Coding Phase is responsible for the actual programming of the modules specified in the Program Design Phase, and the testing of the modules against their specifications.

INPUT

The Coding Phase receives the Implementation Requirements produced by the Program Design Phase. These requirements are: the module specifications, and the intermodule interfaces.

OUTPUT

The coding phase produces coded, tested modules and any necessary documentation.

STEPS

FORMALISM SELECTION

The formalism employed in the coding phase is some kind of programming language. The language may either be selected on the basis of the problem domain and suitability to implementation of the data structures and algorithms specified in the program design phase; or, it may be specified as a constraint from much higher in the design. Examples of programming languages, such as Pascal, PL/I, Algol, or any of a number of assembly level languages, should be familiar to most. One alternative to the conventional programming languages is the preprocessor, which, depending on it and the language it is targeted for, may make coding a much easier and reliable matter through the assistance it can provide in programming style, available control or data structures, data types, concurrency, etc. One of the better known preprocessors is the RATFOR preprocessor for the FORTRAN language [KERN75].

FORMALISM VALIDATION

The formalism validation simply involves examination of the suitability of the selected language with respect to the algorithms, data structures, and general problem domain inherent in the modules to be coded. Other factors are the availability of language processors and program development tools, such as special purpose text editors, and the machine (hardware or software) on which the programs are to run.

EXPLORATION

A one to one mapping between module specifications and coded program modules is performed. The many rules of good programming style and structured program development should be employed.

ELABORATION

On-line syntax checkers and standards enforcers provide valuable assistance in the coding of error-free programs. Library modules are useful in reducing the work of the programmer, provided that care is taken that they actually meet the specifications of the module or module fraction.

CONSISTENCY CHECKING

Consistency checking is usually provided by the language processor or translator, whether it be a compiler, assembler, or interpreter.

VERIFICATION

The coded modules should be verified against the specifications describing them, as passed into the coding phase from the program design phase. The coded modules should then be thoroughly tested. There are tools available for testing, tracing, instrumenting, and performance monitoring.

EVALUATION

The actual performance data can be obtained and checked against the constraints associated with each module. If performance criteria can not be met, then recoding or redesign should be considered.

INFERENCE

The inference step is vacuous for this phase.

INVOCATION

The Coding Phase does not invoke any further phases.

INTEGRATION

Since there are no further phases below the Coding Phase, integrating is vacuous for this phase.

REFERENCES

[ALF077] Alford, M. W., "A Requirements Engineering Methodology for Real Time Processing Requirements," IEEE Trans. on Software Eng. SE-3, No. 1, pp. 60-69, January 1977.

[AMBL77] Ambler, A. L., et al, "Gypsy: a Language for Specification and Implementation of Verifiable Programs," ACM SIGPLAN Notices 12, No. 3, March 1977.

[ANSI66] American National Standard FORTRAN (ANS X3.9-1966), American National Standards Institute, 1966.

[ANSI68] American National Standard COBOL (ANS X3.23-1968), American National Standards Institute, 1968.

[BAKE72] Baker, F. T., "Chief Programmer Team Management of Production Programming," IBM Systems Journal 11, No. 1, pp. 56-73, 1972.

[BAKE73] Baker, F. T., and Mills, H. D., "Chief Programmer Teams," Datamation 19, No. 12, pp. 58-61, December 1973.

[BELL77] Bell, T. E., Bixler, D. C., and Dyer, M. E., "An Extendable Approach to Computer-Aided Software Requirements Engineering," IEEE Trans. on Software Eng. SE-3, No. 1, pp. 49-60, January 1977.

[BOEH76] Boehm, B. W., "Software Engineering," IEEE Trans. on Computers C-25, No. 12, December 1976.

[BRO075] Brooks, F. P., Jr., The Mythical Man-Month, Addison-Wesley, 1975.

[DAHL66] Dahl, O. J., and Nygaard, K., "SIMULA - An Algol-based Simulation Language," CACM 9, No. 9, pp. 671-678, September 1966.

[DAHL70] Dahl, O. J., Myhrhaug, B., and Nygaard, K., "The SIMULA 67 Common Base Language," Pub. S-22, Norwegian Computing Center, Oslo, 1970.

[DAHL74] Dahl, O. J., Dijkstra, E. W., and Hoare, C. A. R., Structured Programming, Academic Press, 1972.

[DIJK68a] Dijkstra, E. W., "Go To Statement Considered Harmful," CACM 11, No. 3, pp. 147-148, March 1968.

[DIJK68b] Dijkstra, E. W., "The Structure of the 'THE' Multiprogramming System," CACM 11, No. 5, May 1968.

[DIJK75] Dijkstra, E. W., "Guarded Commands, Nondeterminacy, and Formal Derivation of Programs," CACM 18, No. 8, pp. 453-457, August 1975.

[DDOD79] Department of Defense High Order Languages Working Group, Preliminary Ada Reference Manual, ACM SIGPLAN Notices 14, No. 6, Part A, June 1979.

[FELD79] Feldman, J. A., "High Level Programming for Distributed Computing," CACM 22, No. 6, pp. 353-368, June 1979.

[GANE79] Gane, C., and Sarson, T., Structured Systems Analysis: Tools and Techniques, Prentice-Hall, 1979.

[HAMI76] Hamilton, M., and Zeldin, S., "HOS - A Methodology for Defining Software," IEEE Trans. on Software Eng. SE-2, No. 3, pp. 9-32, March 1976.

[HETZ75] Hetzel, W. C., editor, Program Test Methods, Prentice-Hall, 1973.

[HOAR69] Hoare, C. A. R., "An Axiomatic Basis for Computer Programming," CACM 12, No. 10, pp. 576-583, October 1969.

[HOAR78] Hoare, C. A. R., "The Engineering of Software: A Startling Contradiction," In Programming Methodology, D. Gries editor, Springer Verlag, pp. 37-41, 1978.

[IBM74] HIPO - A Design and Documentation Technique, GX20-1851, Intl. Business Machine Corp., 1974.

[JACK75] Jackson, M. A., Principles of Program Design, Academic Press, 1975.

[JENS75] Jensen, K., and Wirth, N., PASCAL User's Manual and Report, Springer Verlag, 1975.

[JENS79] Jensen, R. W., and Tonies, C. C., Software Engineering, Prentice-Hall, 1979.

[KERN74] Kernighan, B. W., and Plauger, P. J., The Elements of Programming Style, McGraw-Hill, 1974.

[KERN75] Kernighan, B. W., "RATFOR - A Preprocessor for a Rational Fortran," Software - Practice and Experience 5, No. 4, pp. 395-406, 1975.

[LISK74] Liskov, B., and Zilles, S., "Programming With Abstract Data Types," ACM SIGPLAN Notices 9, No. 4, pp. 50-59, April 1974.

[LISK77a] Liskov, B., et al, "Abstraction Mechanisms in CLU," CACM 20, No. 8, pp. 564-576, August 1977.

[LISK77b] Liskov, B., and Zilles, S., "An Introduction to Formal Specifications of Data Abstractions," In Current Trends in Programming Methodology, Vol. 1, R. Yeh, editor; Prentice-Hall, 1977.

[LISK79] Liskov, B., and Berzins, V., "An Appraisal of Program Specifications," In Research Directions in Software Technology, P. Wegner, editor, MIT Press, pp. 276-301, 1979.

[MYER73] Myers, G. J., "Composite Design: The Design of Modular Programs," Technical Report TR 002406, IBM, Poughkeepsie, NY, 1973.

[NASS73] Nassi, I., and Schneiderman, B., "Flowcharting Techniques for Structured Flowcharting," ACM SIGPLAN Notices 8, No. 8, pp. 12-26, August 1973.

[PAGA81] Pagan, F. G., Formal Specification of Programming Languages, Prentice-Hall, 1981.

[PARN72a] Parnas, D. L., "A Technique for Module Specification, with Examples," CACM 15, No. 5, pp. 330-336, May 1972.

[PARN72b] Parnas, D. L., "On the Criteria to Be Used in Decomposing Systems into Modules," CACM 5, No. 12, pp. 1053-1058, December 1972.

[PARN75] Parnas, D. L., and Siewiorek, D. P., "The Use of the Concept of Transparency in the Design of Hierarchically Structured Systems," CACM 18, No. 7, pp. 401-408, July 1975.

[PARN76] Parnas, D. L., "On the Design and Development of Program Families," IEEE Trans. on Software Eng. SE-2, No. 3, pp. 1-9, March 1976.

[RIDD78] Riddle, W. E., et al, "Behavior Modeling During Software Design," IEEE Trans. on Software Eng. SE-4, No. 4, pp. 671-678, July 1978.

[ROBI77] Robinson, L., and Levit, K. N., "Proof Techniques for Hierarchically Structured Programs," CACM 20, No. 4, pp. 271-283, April 1977.

[ROSS77a] Ross, D. T., and Schoman, D. E., "Structural Analysis for Requirements Definition," IEEE Trans. on Software Eng. SE-3, No. 1, pp. 6-15, January 1977.

[ROSS77b] Ross, D. T., "Structured Analysis (SA): A Language for Communicating Ideas," IEEE Trans. on Software Eng. SE-3, No. 1, pp. 16-34, January 1977.

[TEIC77] Teichroew, D., and Hershey, E. A., "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems," IEEE Trans. on Software Eng. SE-3, No. 1, pp. 41-48, January 1977.

[WEGB76] Wegbreit, B., "Verifying Program Performance," JACM 23, NO. 4, pp. 691-699, October 1976.

[WEGN79] Wegner, P., Research Directions in Software Technology,
MIT Press, 1979.

[WIRT71a] Wirth, N., "The Programming Language PASCAL," Acta Informatica
1, No. 1, pp. 35-63, 1971.

[WIRT71b] Wirth, N., "Program Development by Stepwise Refinement," CACM
14, No. 4, pp. 221-227, April 1971.

[WIRT73] Wirth, N., "On the Composition of Well-Structured Programs,"
Computing Surveys 6, No. 4, pp. 247-259, December 1974.

[WULF76] Wulf, W. A., London, R. I., and Shaw, M., "An Introduction to
the Construction and Verification of Alphard Programs," IEEE
Trans. on Software Eng. SE-2, No. 4, pp. 243-265, December 1976.

[YOUR75] Yourdon, E., Techniques of Program Structure and Design,
Prentice-Hall, 1975.

2.3.4 MACHINE DESIGN STAGE

TERMINOLOGY OF THIS STAGE

REQUIREMENTS: Hardware Requirements
PHASE: Hardware Configuration Design
REQUIREMENTS: Component Requirements
PHASE: Component Design
REQUIREMENTS: Circuit Design Requirements; Firmware Requirements

DESCRIPTION OF STAGE ACTIVITIES

This stage receives as input a set of hardware requirements, which may consist of a set of procurement instructions and/or design specifications for customized hardware (such as the desired instruction set and performance constraints, or some form of signal transformation function). The purpose of this stage is to procure hardware for the system and to carry out a high level design of all custom hardware. The output of this stage, if necessary, is a set of firmware requirements for the hardware that has been purchased or designed and a register level specification of the hardware circuitry which must be custom made.

STATE-OF-THE-ART

Well established methodologies already exist in many places for the development and procurement of hardware at this level, but most suffer from serious shortcomings. Despite the wealth of knowledge and experience that has been developed over the years of large hardware systems design, the design of these system remains an art. Much of this is due to a lack of formality in the specification methods used and the lack of a cohesive support facility for the development of hardware. In recent years, however, the development of hardware description languages [SHIV79] and efforts by the American National Standards Institute to establish a standard formal symbology for hardware description have paved the way for the establishment of hardware design facilities meeting the requirements of the TSD framework.

PHASE NAME: Hardware Configuration Design

PURPOSE

This phase serves to procure off-the-shelf hardware to meet the system hardware specifications where possible and to carry out an architectural design of the hardware which cannot be purchased.

INPUT

The Hardware Requirements specification consists of one or more of the following :

- A set of procurement specifications for the purchase of existing hardware systems.
- A set of specifications for the design of custom hardware including the required instruction set or processing capabilities and the required performance of the hardware system.
- A set of transformation functions for hardware which must provide an interface with the environment.

OUTPUT

The output of this phase is the Component Design Requirements specification, which is a formal model of the system at the hardware architecture level (the building blocks for the model at this level are processors, memories, switching networks, interconnection links, etc.). This model must include both a functional and performance model of the system, with orientation toward the implementation of the instruction set presented in the input requirements in such a way that all of the constraints are met.

STEPS

FORMALISM SELECTION

The formalism used in this phase must be designed to work with design components at the level of processors, memories, and communications structures. For those portions of the system which must be custom designed, the formalism should be oriented toward the type of machine architecture being designed (a formalism designed for use in distributed systems design would be more complex than is needed for the design of a single stand-alone computer system). If procurement is the desired goal, then the formalism should be oriented toward the evaluation of existing machines in terms of the Hardware Requirements. In all cases the formalism should allow for the description and analysis of system constraints as well as function. (See [BELL71] for

one formalism which is widely used at this level.) Such factors as human engineering and availability of automated aids should be considered when evaluating any formalism for potential use.

FORMALISM VALIDATION

Validation of the formalism chosen consists of performing an analysis of suitability of the formalism for use with the particular design and procurement problems presented by the proposed hardware system. In many cases this is done by using the formalism to construct a "toy" model of the proposed system and then evaluating the formalism in terms of this model.

EXPLORATION

The exploration task consists of two possibly interrelated tasks: the search for existing hardware systems for purchase, and the design of the hardware architecture for custom machinery. The search for existing equipment essentially involves a survey of current commercial hardware, interviews with manufacturer representatives, analysis of established customer installations, etc. In the design of custom hardware, a modular approach to design emphasizing flexible and expandable structures with simple interfaces is required. In all cases a review and incorporation of past efforts in similar areas could greatly reduce the amount of effort necessary in this step.

ELABORATION

In this step the results obtained in the previous step are expressed formally using the formalism chosen for this phase. In this way the design of custom components and the characteristics of proposed off-the-shelf hardware is put in a form which allows extensive analysis in later steps. Because of the complexity and formality required, some form of computer aid is needed to support this activity. Examples of this are storage of the specification in a database and automatic generation of documents from the database, syntax checking on the specification, maintenance of a library of standard components and solutions to specific problems, and the handling of simple clerical chores for the designer [SHIV79].

CONSISTENCY CHECKING

This step, which should be carried out with as much automated support as is possible, serves to analyze the formal specification of the hardware system for correct usage and self-consistency. No attempt is made at this step to analyze the system itself, however. Examples of inconsistent specifications are: requiring communication between machines of different word lengths or processing speeds without an interface, specifying a processor component without providing any memory for it, specification of a communications link without a

termination, etc. Automated support for this could take the form of a set of analysis programs to check the specification for specific consistency criteria or some kind of formatting system to present the specification in a form that can be easily analyzed by the designer.

VERIFICATION

In this step the hardware system itself is analyzed functionally for correctness with respect to the Hardware Requirements. In particular, sequencing and protocol in the system must be verified, as well as the support for all of the required hardware functions such as the hardware instruction set and input/output functions. Mechanical analysis here may be difficult as it requires comparison of two (perhaps completely different) formal specifications for mutual consistency, but some kind of computer aid to format the specification in a form that is convenient for the comparison of the two specifications would be helpful. In addition, the function of processor components in the system may be verified by emulation of their instruction sets [CLAR78].

EVALUATION

The purpose of this step is to evaluate the design in terms of the constraints presented in the Hardware Requirements. The performance and timing of the system can be evaluated in part by simulation of the system [TO74], although in cases which are simple or very regular in structure some form of analytic technique may be developed [ALLE80]. For devices which have been purchased, a set of benchmark tests designed to evaluate the hardware for the specific task that it will perform may be used to evaluate the equipment. For processor type components, emulation may be used in conjunction with estimates of the required execution time for each instruction to gather performance data. Other constraints such as power and weight limitations may be measured directly for existing equipment and must be estimated for custom equipment. Aspects such as fault tolerance and maintainability may also be of concern in this step [COX79].

INFERENCE

This step attempts to project the impact that decisions made at this level of the design will have on any later phases of the design and on later use of the system in a production or maintenance environment. Such a projection is to serve both as a guide to later design phases and as a final analysis of issues other than functionality and performance that may have an impact on the acceptance of the design. Examples of issues that are of concern in this step are the feasibility of construction of the hardware as specified, manufacturer support for procured hardware, availability of off-the-shelf parts for implementation of the next level of design, manpower requirements to complete the system construction, determination of critical areas which require extra design effort (such

as processing bottlenecks in a high performance system), etc.

INVOCATION

This step invokes the Component Design Phase. If all of the necessary hardware, complete with firmware, has been procured then this step is omitted.

INTEGRATION

In this step all of the hardware (both procured and custom designed) is integrated and tested. The major task of the step is to insure that the hardware system as a whole works as it was designed before releasing it for integration with the system software. Benchmark programs and simulation of the projected operating environment are two methods which may be used to isolate errors and confirm correct operation [T074].

PHASE NAME: Component Design

PURPOSE

The purpose of this phase is to analyze the component requirements, purchase all commercially available components (processors, memories, I/O controllers, etc.) which meet the component requirements and carry out a register level design of those components which must be custom built. In addition, the requirements for system firmware must be determined in this phase.

INPUT

The input to this phase is the Component Requirements specification, which is a description of the architectural components comprising the hardware system. This includes both a functional specification and a set of constraints (speed, size, etc.) over these components. An important part of this specification is the intended instruction set for the processor elements with its requisite timing.

OUTPUT

There are two sets of output requirements for this phase: the Circuit Design Requirements and the Firmware Design Requirements. The Circuit Design Requirements specification is a register level description of the design of those components which cannot be bought commercially. This specification must include both the functionality of the register level circuits used as primitives and a description of the speed, size, and power constraints for the circuit. The Firmware Design Requirements is a specification of the functionality, performance, and memory constraints required of the system firmware. This is typically given in terms of an instruction set to be implemented given the register structure of the hardware, system protocols to be established, and constraints over timing and memory usage for these functions.

STEPS

FORMALISM SELECTION

The range of activities to be carried out in this phase is fairly large, and a variety of formalisms may need to be selected here in order to carry out the activities of this phase. One of the primary activities of this phase is the procurement of commercial hardware components such as processors, memories, mass storage, communications interfaces, etc.; and this is aided by a formalism oriented towards the description of these components in a formal manner that lends itself to later analysis (such a general formalism does not currently exist). The second major activity is the further design of those components which cannot be purchased. The formalism used for this must support hardware description at the register transfer level (typical

primitives at this level are storage and shift registers, ALUs, bus structures and controllers, multiplexers, etc.) and be able to model system constraints as well as function in an analyzable manner. The last major activity is the establishment of the firmware requirements for processor-type components. The formalism used here must be capable of modelling the register level architecture of the hardware to be microprogrammed as well as present a functional and performance model for that firmware. Each of these formalisms should be compatible as the analysis and evaluation of the system requires coordinated analysis of each of these specifications. Some examples of formalisms possessing some of the required properties are SMITE, ISPS, CDL, and DDL [SHIV79].

FORMALISM VALIDATION

The basic requirement in this step is that the formalisms selected be able to support the types of analysis and descriptive tasks required of them. The use of the formalisms on simple test cases and experience from any previous use of the formalisms should provide the basis for establishing the validity of using the formalism for any particular design. Human factors such as usability should be considered when validating a formalism.

EXPLORATION

It is in this step that the major design activities of the phase are carried out. A review of commercial components must be made, and a study of their potential use as components in the design must be made. Those components which cannot be purchased must be custom designed, and this design must be carried out to a register transfer level. If a microprogrammed control structure is to be used, a set of microinstructions must be decided upon and implemented in the control structure design. The microinstruction set along with the requirements for the machine instruction set and timing establish the basic requirements for the system firmware. As always, the principles of good design emphasizing modularity, simple module interfaces and interconnections, maintainability, etc. should be followed.

ELABORATION

This step serves to express the informal designs, firmware requirements, and commercial component descriptions in the formalisms selected for this phase. The relative simplicity of the basic primitives at this level should allow for a great deal of structuring in the formalisms, which in turn should enable mechanized support for building these models. This support will usually take the form of computer aided construction of a design database, syntax checking on all input constructs to this database, prompting for missing components of the specification, and specially formatted output of the specification for designer verification [SHIV79].

CONSISTENCY CHECKING

The purpose of this step is to evaluate the formal models produced above for self consistency and for consistency with each other. Examples of inconsistent specifications are microcode instructions which act on registers not defined in the register description of the processor, connecting byte-parallel devices with bit-serial controllers, or attempts to place 36-bit data values on a bus which has only 32 data lines. Again, due to the relative simplicity and structure at this level it should be possible to automate a great deal of this consistency checking through a series of analysis programs which act on the specification database.

VERIFICATION

In this step the design of the system is analyzed for functional correctness with respect to the Component Requirements. For small systems this may be done manually by walking through the sequences of events which occur in the system. For larger systems, automated tools such as simulation may be used. Protocols between the custom components and the purchased components should be analyzed for correctness. The microcode instruction set should also be analysed in terms of the purchased and custom components to insure that the microinstructions are all implemented. Procured components may also be tested in simulated operating environments to verify their functionality.

EVALUATION

In this step a functionally correct design is analyzed for conformance with the performance requirements and other constraints established for the system. Items to be considered at this point are projected speed, projected size, projected weight, etc. Most of these constraints can be measured directly with the purchased hardware. For the custom hardware, emulation of the microcode level and simulation of the circuit activity using estimates of the timing of these elements can be used to gather performance data. Additional analysis that should be performed at this step includes fault tolerance analysis, failure rate predictions, and probability of entering a deadlock state. In all cases some kind of automated aid to set up the component tests and simulations would greatly simplify the task of this step [TO74, COX79].

INFERENCE

The purpose of this step is to analyze the design decisions made in this phase in terms of their impact on the later phases of the design. In particular, items such as the feasibility of implementation of the design, circuit implementation technology, strategies for firmware design, effects on system maintainability, and effects on system usability all need to be considered. The conclusions reached

here serve as a guide to designers at the next levels and to identify areas in the design which will require extra attention at the next levels (such as a high speed ALU and efficient firmware to drive it).

INVOCATION

The phases invoked at this point are the Circuit Design Phase and the Firmware Design Phase.

INTEGRATION

This step serves to integrate the individual circuits designed in the circuit design stage into components, and integrate the firmware developed in the firmware design stage with the custom and commercial components considered in this phase. The components must then be individually tested and debugged, usually through some sort of simulation of the projected operating environment. Only after these components have been shown to meet the functional and performance requirements established for them are they turned over to the hardware configuration design phase for further integration.

REFERENCES

- [ALLE80] Allen, A. O., "Queueing Models of Computer Systems," Computer, pp. 13-24, April 1980.
- [BELL71] Bell, C. G. and Newell, A., Computer Structures: Readings and Examples, McGraw-Hill, 1971.
- [CLAN78] Clark, N. B., "Common Software Support Environment," White Paper, Rome Air Development Center, Rome, New York, 1978.
- [COX79] Cox, L. A., Jr., "Performance Prediction from a Computer Hardware Description," Report Number NPS52-79-001, Naval Postgraduate School, Monterey, CA, January, 1980.
- [SHIV79] Shiva, S. G., "Computer Hardware Description Languages - a Tutorial," Proceedings of the IEEE 67, No. 12, pp. 1605-1615, December 1979.
- [TO74] To, K. and Tulloss, R. E., "Automatic Test Systems," IEEE Spectrum pp. 44-52, September 1974.

2.3.5 CIRCUIT DESIGN STAGE

TERMINOLOGY OF THIS STAGE

<u>REQUIREMENTS:</u>	Circuit Design Requirements
<u>PHASE:</u>	Switching Circuit Design
<u>REQUIREMENTS:</u>	Electrical Circuit Requirements
<u>PHASE:</u>	Electrical Circuit Design
<u>REQUIREMENTS:</u>	Solid State Requirements
<u>PHASE:</u>	Solid State Design
<u>REQUIREMENTS:</u>	Fabrication Requirements
<u>PHASE:</u>	Fabrication

DESCRIPTION OF STAGE ACTIVITIES

The CIRCUIT DESIGN STAGE is comprised of four phases called the switching circuit design, electrical circuit design, solid state design, and fabrication. These phases will be treated summarily as the principles and techniques used in these phases are already well understood and contribute little to the understanding of the TSD philosophy as a whole. Because of the shortened treatment, the format of this section will not follow that of the other stage descriptions.

The SWITCHING CIRCUIT DESIGN phase deals with the design of custom circuits at the level of logic functions, op-amps, A/D converters, flip/flops, etc. The input to this phase is the Circuit Design Requirements specification, which is a register transfer level description of the circuit with timing and physical constraints included. The objective is to produce a full description of the circuit at the logic gate level, and to obtain off-the-shelf circuits when performance, size, power, and other constraints allow. Some formalisms which are often used in this phase are Boolean Algebra and switching theoretic techniques, augmented with performance models for logic circuits under current technologies. Analog components require specifications such as transfer functions or Bode plots. Exploration includes a review of current commercial circuit packages and techniques for constructing circuits within present technology, as well as systematic design of custom circuitry. Procurement, as usual, is preferred at this stage if all functional requirements and constraints can be met. Evaluations of the circuits at this level consist typically of analytic methods based on graph theory and switching theory, simulation of the circuit action, and breadboarding. The input to the next phase is the Electrical Circuit Design Requirements, consisting of logic and analog circuit models of the system augmented with performance requirements and other physical constraints. After invocation of this phase, the custom designed circuits and the off-the-shelf circuitry must be integrated and tested, usually through a small scale simulation of the target environment.

The ELECTRICAL CIRCUIT DESIGN phase deals with the circuit design at the level of conceptual devices such as transistors, resistors, capacitors, etc. The input to this phase is the Electrical Circuit Design Requirements, which is typically a logic circuit model with added

performance and other constraints. The objective of this phase is to transform these requirements into a design composed entirely of conceptual electrical devices. The usual formalism used at this level is the standard electrical schematic diagram, but the ability to model device characteristics and circuit layout geometry must be present at this level. It is at this level that the characteristics of the physical medium of implementation of the circuit begin to have a great effect on the design, principally through the type of devices available to the designer but also in such considerations as loading of circuits, amplifier gains, transmission line effects, etc. Hence, the technology used to implement the circuit must be decided at this phase. The exploration and elaboration steps rely largely on a review of existing circuit construction techniques within each technological family, and the principles of design at this level are comparatively well understood if not always followed. Although for small circuits or one-time productions the procurement of discrete transistors, inductors, etc. for circuit construction is preferred, the decreasing startup costs of integrated circuit fabrication may make implementation via integrated circuit more practical as time goes on. Evaluation of a design at this level is largely done through manual analysis and breadboarding, although computer simulation of circuits at this level is available (and expensive). The output of this phase is the Solid State Design Requirements, which is usually a specification of the electrical devices comprising the circuit and a set of physical constraints and assumptions concerning these devices. Once these devices have been procured or fabricated, they are assembled and tested in the integration step, typically by simulation of the operating environment or application of a set of test signals designed to put the circuit through a significant portion of its active state set.

The SOLID STATE DESIGN phase serves to translate the electrical circuit specification into a form suitable for fabrication. The input to this phase is the Solid State Design Requirements, which contains information concerning the device interconnections and device characteristics of the integrated circuit to be constructed, along with the assumed fabrication technology. The formalisms used in this phase are mainly graphical, and in most places are aided extensively by computer. It is at this point that all device dimensions are fixed, interconnection patterns and chip geometry worked out, and approximate physical and performance characteristics of the end circuit determined. To aid in this process, many facilities have developed a set of standard design rules for each technology, which if followed will guarantee a chip design that can be fabricated successfully. The exploration and elaboration steps themselves involve a great deal of exploration into possible circuit structures and layouts, but a fairly large body of existing solutions to layout problems is making this task easier. Evaluation is done typically through analysis for conformance with these design rules, and may also include automated analysis for conformance with performance requirements. The output of this phase is the Fabrication Requirements specification, which is some form of formal specification of the geometry and layout of the chip to be fabricated. The integration step here is nonexistent, as the circuit arrives in integrated form from the fabrication phase.

AD-A126 101

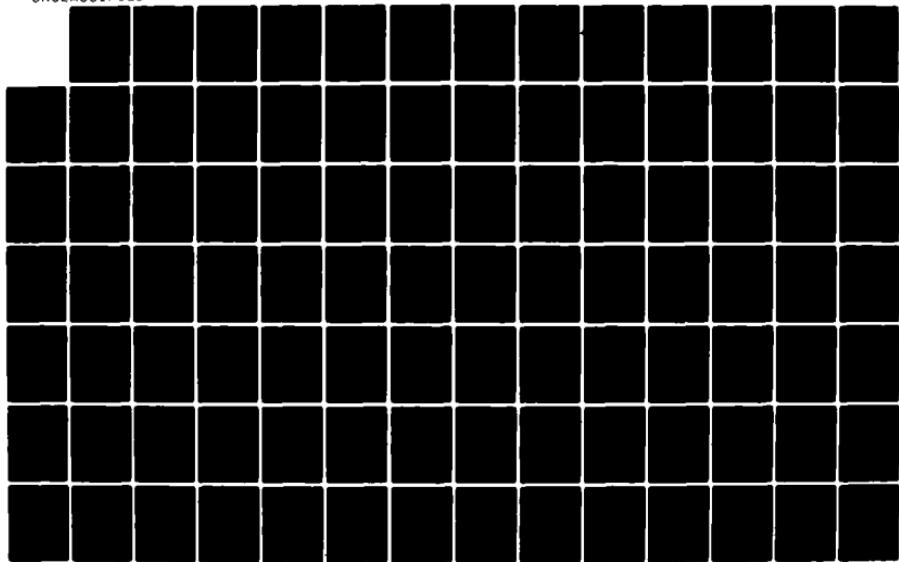
TOTAL SYSTEM DESIGN (TSD) METHODOLOGY ASSESSMENT(U)
WASHINGTON UNIV SEATTLE DEPT OF COMPUTER SCIENCE
G ROMAN ET AL. JAN 83 RADC-TR-82-331 F30602-80-C-0284

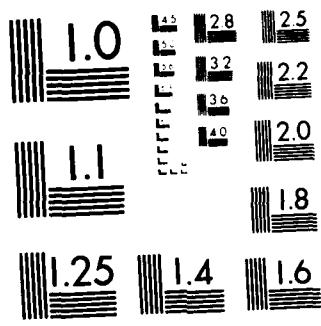
3/4

F/G 9/2

NL

UNCLASSIFIED





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963 A

The FABRICATION phase serves to translate the geometric specification of the integrated circuit to be produced into a finished circuit. The input to this phase is the Fabrication Requirements specification, described above. The formalism used is the specific fabrication process for the device, which involves the generation of process masks from the input specifications and the use of these masks to guide the fabrication process. Exploration and elaboration are automatic at this point and involve simply the production of the circuit. Verification of the design at this level takes the form of visually checking for flaws in the fabrication of the circuit and checking the operation of embedded test circuits, and evaluation takes the form of electrically testing the circuit. The output of this phase is a finished circuit.

2.3.6 FIRMWARE DESIGN STAGE

TERMINOLOGY OF THIS STAGE

REQUIREMENTS: Firmware Requirements
PHASE: Microcode Design
REQUIREMENTS: Microcode Requirements
PHASE: Microprogramming
REQUIREMENTS: Microcode Generation Requirements
PHASE: Microcode Generation

DESCRIPTION OF STAGE ACTIVITIES

The purpose of the firmware design stage is to translate the firmware requirements into executable microcode, along with appropriate documentation and analyses. The complexity of application in which firmware is useful ranges from very large systems, such as a PASCAL engine, through less complex applications, such as a graphics display module or a matrix multiplication module, to relatively simple applications, such as the implementation of specific machine language instructions. Firmware development is similar in many ways to general software development, and many of the concepts, techniques, and tools applicable in that discipline also are useful in the development of firmware. However, there are a number of ways in which general software and firmware differ; because of these differences, techniques not normally applied in the development of software are sometimes required for the development of effective firmware. Firmware often deals with more complex, detailed and low-level hardware components (and the data paths between them) than does software. This, combined with parallel manipulation of the hardware components and data paths, requires unique code generation, verification, and optimization (often known as compaction) techniques not used in software development. The input to this stage, firmware requirements, includes:

Functional Specification

-- The semantics of the action to be performed are usually specified by use of a register transfer language (RTL). The objects manipulated by this specification are only those defined by the user (person who will utilize the firmware) and do not include the actual hardware components available for computation.

Constraints

-- The pertinent aspects of the hardware architecture (e.g., horizontal or vertical) must be specified; this includes the logical and timing properties of the hardware components and their interfaces. A specification of the microinstructions must be given, detailing the format, semantics, and timing of each microinstruction. Requirements on performance (i.e., timing) and space limitations also must be specified.

The output of this stage is:

- The microcode itself.
- Documentation of the microcode.
- Any analyses (such as timing summaries) that may be useful for confirmation at higher levels of the total development process.

The firmware design stage is divided into three phases: the microcode design phase, the microprogramming phase, and the microcode generation phase. In performing the activities in these phases, four languages will be postulated. L1 is the primary functional specification language in which the semantics of the firmware requirements are expressed (in terms of the user-defined registers). L2 is a secondary functional specification language in which the semantics can be expressed in terms of the actual hardware components and data paths to be used by the eventual microcode developed. Two major alternatives for L2 are an RTL and a high-level microprogramming language (HML). L3 is an implementation language in which the details of the semantics and structure of the final microcode can be expressed. The major alternatives for L3 are an HML and an assembly language. L4 is the microcode itself.

In the microcode design phase, L2 is selected. First, a microprogram design activity is done, in which common subfunctions are identified and functionally complete computational components are associated with self-contained microprogram modules. During this design, specific algorithms for accomplishing the required tasks are chosen; these algorithms must be proven to be correct and documented. Then the functional specification of the firmware requirements as expressed in L1 are translated into L2. During the translation, the structure of the microprogram design is incorporated and design decisions are made about which hardware components and data paths are to be used to accomplish the required tasks. The semantics as expressed in L2 become the functional specification for the next phase. The performance constraints of the firmware requirements are distributed to the microprogram modules identified in the microprogram design activity; these become the performance constraints of the next phase.

In the microprogramming phase, L3 is selected, and the semantics as expressed in L2 are translated into L3. During the translation, certain implementation decisions are made, such as subroutines vs. functions, global vs. local parameter passing, and temporary handling. The semantics as expressed in L3 become the functional specification for the next phase. The performance constraints of this phase, with virtually no modification, become the performance constraints of the next phase.

In the microcode generation phase, the semantics expressed in L3 are translated into L4, the microcode itself. During the translation, optimization of the final microcode is performed. The means of performing the translation and the type of optimization to be performed must be determined. Although testing and integration are performed in all three phases, the majority of the microcode testing is done in this phase.

The intent is to produce methodologies in which the maximum amount of computer automation that can be incorporated is actually utilized. To this end, we advocate methodologies in which the specification languages, L1 and L2, are as formal as possible. This facilitates machine manipulation of the specification text, allows the utilization of automated design aids for consistency-checking and verification, and encourages the development of automated or semi-automated design aids for translating between the various languages. Throughout this section, both automated and unautomated techniques will be discussed as possible options; this reflects the current state of the art. However, the emphasis is on the development of automated design tools and their incorporation into an integrated, coherent system in which the effort expended by a human designer in the process of developing a final product is minimized.

Two major observations about the effort expended in the phases of this stage should be made clear. First, L2 and L3 should be selected to be as similar to one another as possible; this will reduce the effort in the microprogramming phase. In fact, it may be possible to select L2 to be the same as L3 (or a subset thereof). An obvious choice for such a combined L2-L3 language is a high-level microprogramming language (i.e., L3) capable of expressing the semantics (of L1) in terms of the hardware components and data paths available to the microcode (i.e., L2). The utilization of such a combined language would make the microprogramming phase essentially vacuous, although it would tend to push some implementation decisions into the design phase. Second, if L3 is selected to be a high-level language and various optimization tools are integrated into its implementation, then the microcode generation phase becomes essentially vacuous. These two observations, along with the fact that high-level language programs are more easily produced and modified than low-level language programs, are a strong motivation for selecting L3 to be a high-level microprogramming language.

STATE-OF-THE-ART

In order to understand the objective of introducing firmware into the total system design, a general overview of the properties, uses and intent of firmware seems appropriate. There are usually three alternatives for the implementation of any specific function: software, firmware, and hardware. Hardware is fastest in execution, but is not flexible (when simple corrective modifications must be made) and may require a significant design effort. Software is slowest in execution, but is very flexible and requires less design effort. Firmware is a compromise which combines the flexibility and speed of design of software with some of the execution speed of hardware. Thus, firmware is most applicable when a flexible, relatively fast system must be developed with a relatively small design effort.

The execution speed of the final system is very important. The speedup (of firmware over software) is accomplished due to the inherent speed at which microinstructions can be executed and the potential parallelism of accessing many hardware components at the same time. (Reduction in storage space requirements also may be realizable by the use

of microcode as compared with macrocode, but this usually is secondary to the speedup advantage.) Since one of the intents of introducing firmware into the system design is to achieve this execution speedup, an obvious objective is to produce firmware that contains a minimal number of microinstructions and can produce the desired semantic actions in the shortest possible amount of time. Thus, the results (and therefore the quality) of the optimization activity is important in the development of effective firmware.

There are two commonly utilized characterizations for firmware architectures: vertical and horizontal. A vertical architecture is one in which each microinstruction incorporates little or no parallelism in the manipulation of different hardware components. A horizontal architecture is one in which each microinstruction has the capability to manipulate a significant number of hardware components in parallel. Most of the firmware systems used in practice combine these two architectures into a mixed implementation. Speedup can be obtained in vertical architectures, but this speedup is limited by the small degree of parallelism. Local [AH076, BUSA69, FRAI70] and global [ALLE70, ALLE76, COCK70, EARN72, GILL77, GRAH75, HECH72, HECH73, HECH74, HOPC72, KAM76, KENN71, KENN75, KILD73, OSTE74, SCHA73, ULLM72] optimization techniques used in general software development are normally sufficient to produce reasonably optimized code for such a vertical architecture. On the other hand, very significant speedup can be realized in horizontal architectures by taking advantage of the inherent parallelism. However, optimization techniques not used in general software development (because the inherent parallelism is not present) must be used to produce significantly optimized code. The introduction of parallelism also makes it more difficult to verify that the action of the optimized code represents the semantics of the original specifications.

The selection of L3 can have a significant affect on the total effort. It should be clear that developing a program (whether software or firmware) in a high-level language requires less effort to code originally and maintain than one developed in a low-level language, such as assembly language. However, this is not the only advantage of writing in a high-level language. Optimization tools can be incorporated into the underlying implementation of the high-level language to perform automatic optimization of the resulting code. Also, it is possible to incorporate an assertion sublanguage within the high-level language to aid in automatic verification of the resulting code. This is not intended to imply that such tools cannot be incorporated into low-level languages; however, the natural way in which they can be packaged in a high-level language, the ease of use of the high-level language, and the way in which the high-level language releases the programmer from burdensome details all indicate that this is the more appropriate course. A high-level language also may be applicable to more than one firmware system.

It is a common belief that hand optimization of microcode always produces more effective firmware than machine optimized code. Experience has shown that, for small microcode segments, this probably is true. However, experiments on large firmware systems indicate that machine optimized microcode can produce better firmware than hand optimized microcode [PATT79]. This may be due to the observation that for small

segments of microcode, the complexity is small enough that a person performing hand optimization can do an excellent job of optimization; however, as the size of the microcode (and therefore its complexity) increases, the person is generally unable to effectively manage the increased complexity, and thus fails to recognize optimizations that an automated optimizer is capable of recognizing.

A significant effort has been expended to develop high-level microprogramming languages for microprogrammable machines (such as SMITE [SMIT77]). Work has been done to develop similar languages that are applicable to a wide variety of machines instead of just one. Progress in this area has been hampered by the facts that: many of the microprogrammable machines have a fundamentally different detailed architecture, the semantics of certain microinstructions of one machine may have no corresponding counterpart in another machine, and the development of effective firmware requires utilization of these specialized architectures and microinstructions. No specific approach to these problems has shown itself to be the ultimate solution. However, two specific approaches seem promising. The first approach uses the concept of extensibility, such as EMPL as proposed by DeWitt [DEWI76a, DEWI76b]. In such an approach, a core language is defined in which the properties common to most machines are explicitly expressible (e.g., control structures, data transfer). The language is designed so that extensions to the syntax and semantics can be introduced into the language by use of appropriate directives contained in the core language. In this way, such a high-level language can express those aspects common to most machines (by use of the core language), and the differences can be expressed by use of extensions. A second approach is similar in intent to the extensible language approach but uses a machine-independent microprogram language schema to accomplish the goal [DASG78a, DASG80a, DASG80b]. In this approach, the language is defined at several different levels. The top level constitutes a core language (similar to that of extensible languages) in which aspects common to most machines can be expressed. Lower levels define more detailed aspects of the machine; these lower levels are 'instantiated' for the specific machine upon which the microcode will execute. In other words, the differences between two specific machines become apparent and are introduced only at the lowest level necessary. Thus, significant amounts of code will be applicable to a wide variety of machines. One or both of these approaches eventually may produce a language (or family of languages) applicable to a wide range of microprogrammable machines.

An area in which essentially no work has been done is that of mapping the original firmware requirements onto the specific hardware architecture of the machine. This activity is currently done primarily by hand and is presumed to require the skill of a highly trained designer. If a coherent, integrated, automated facility is to become a reality, the development of an automated or semi-automated design tool to help in this activity is essential.

PHASE NAME: Microcode Design

PURPOSE

The purpose of the microcode design phase is to produce and document an overall design of the microcode. This includes selection of a specific program structure and algorithms for performing various functions (such as a multiplication or table lookup).

INPUT

The firmware requirements include:

- Functional specification of the firmware.
- Hardware description (components, data paths, and communication).
- Microinstructions specification (format, semantics).
- Performance constraints.

OUTPUT

The microcode requirements include:

- Microprogram organization.
- Module specifications.
- Hardware Description (same as in input).
- Microinstructions specification (same as in input).
- Performance constraints (time constraints for each module).

STEPS

FORMALISM SELECTION

The microcode design language, L2, must be selected. The selection criterion for the language is a function of the hardware components, their data paths, and their forms of communication; L2 must be able to express the original functional specification in terms of the actual hardware the final microcode will manipulate. L2 should be as high-level as possible (suppressing certain implementation details) but should be as similar to L3, the implementation language, as possible (to reduce translation effort in the next phase).

The selection of L2 may, of course, depend on the specific tools available. It is usually chosen to be an RTL or a subset of an HML. Since automation is one of the primary objectives, its specification form should be machine manipulatable and its semantics should be formal enough that at least certain forms of verification can be performed upon it.

FORMALISM VALIDATION

Validation is normally performed by hand by a designer familiar with the firmware architecture. This can be done by exhaustive verification that each of the hardware components, data paths and communication primitives (in all their combinations) can be expressed within the formalism of L2.

EXPLORATION

The exploration step embodies two basic activities which may be intertwined. The first is microprogram design in which common subfunctions are identified and functionally complete computational components are associated with self-contained microprogram modules. Specific algorithms must be selected to perform various tasks. These algorithms must be well documented and proven correct with respect to the environment in which they will operate. The concepts and techniques used here are well understood and are very similar to those used in software program design. The second deals with mapping the original functional specifications (as expressed in L1) into specifications that address the actual hardware to be manipulated (L2). This activity is viewed as being an art and requiring skilled personnel familiar with firmware design.

ELABORATION

The elaboration step deals with incorporating the decisions made in the exploration step within the process of translating the functional specifications from L1 to L2. There are no known tools available to help in this translation process. However, the development of such a design aid is very important for the development of an integrated design facility, and a significant effort should be expended in this effort.

The results of this elaboration step become the module specification input for the next phase.

CONSISTENCY CHECKING

Besides eliminating syntax errors, the consistency checking step deals with verifying that the semantics expressed in L2 are consistent as a function of the hardware. For instance, no two parallel data transfers should utilize the same data path; nor should two parallel data transfers have a common target. If L2 has been chosen to be an HML, such consistency checking may be an integral part of the implementation of the corresponding compiler. In any event, if such consistency checking functions are not available elsewhere, specific

design aids should be developed. Of course, such design aids must have some knowledge of the hardware (at least local semantics), and the effort required to develop them will depend on the specific hardware (or class of hardware) involved.

VERIFICATION

The verification step deals with ensuring that the specifications expressed in L2 are consistent with those expressed in L1. Since the design activity constitutes a significant translation between (potentially) two distinctly different domains, it may be difficult to develop design aids that can formally guarantee consistency between the specifications. Such a design aid would require a very detailed knowledge of the hardware (such as timing). It might involve some form of global simulation; a concept similar to symbolic execution might be employed.

EVALUATION

The evaluation step deals with distributing the performance constraints of the firmware requirements onto the microprogram modules identified in the exploration step. This distribution is based on the specific decomposition chosen and the designer's judgements (estimates) about the actual properties of the microcode to be produced. Although design aids could be developed to help distribute the affect of these judgements onto the separate modules, this is usually easily done by hand, and such aids seem unwarranted.

INFERENCE

With a specific design developed in this phase, it may be impossible to implement (in the next two phases) the firmware given the space and execution time constraints. If it is determined that the specific design cannot be effectively implemented, then a new design must be developed (or failure reported to the invoking stage).

INVOCATION

The microprogramming phase is invoked. This may be done separately for each distinct module, certain collections of modules, or the entire microcode design.

INTEGRATION

This step deals with integrating separate microcode modules together to verify that the total firmware package works as an integrated whole. For instance, it must be verified that all the modules have consistent interfaces. A systematic testing strategy should be developed to insure that the firmware is consistent with the original functional specifications. It must also be verified that the actual microcode meets the input performance specifications. If the actual hardware is available, this testing can be performed on it; otherwise, the hardware may be emulated (or simulated).

PHASE NAME: Microprogramming

PURPOSE

The purpose of the microprogramming phase is to translate the results of the microcode design phase into an implementation language capable of producing actual microcode.

INPUT

The microcode requirements include:

- Microprogram organization.
- Module specifications.
- Hardware description.
- Microinstructions specification.
- Performance constraints.

OUTPUT

The microcode generation requirements include:

- Implementation specifications (HML or low-level assembly language).
- Hardware description (same as the input).
- Microinstructions specification (same as the input).
- Performance constraints (usually the same as the input).

STEPS

FORMALISM SELECTION

The implementation language, L3, must be selected. It must be able to express the semantics of the actual microcode to be executed. The selection criteria include: the specific tools available, ease of semantic expression, ease of translation from L2 to L3, and foresight about optimization, modifiability, and verifiability. There must be an effective translation process from L3 to the microinstructions. The major choices for L3 are an HML and a low-level assembly language.

FORMALISM VALIDATION

The validation criteria are similar to that of the microcode design phase. If L3 is chosen to be an assembly language, this step is vacuous; if a HML is chosen, an exhaustive verification can be performed. The appropriateness of the language chosen is based on the criteria mentioned in the formalism selection step and the quality of the actual microcode produced. For instance, assembly language may be difficult to optimize and verify, whereas an HML may not produce sufficiently compacted code.

EXPLORATION

The implementation decisions that must be made are very similar to those found in software implementation. Such decisions include: whether to implement a module as a subroutine or a function, whether to pass data globally or explicitly through a parameter list, and how to hold temporary values.

ELABORATION

The specification expressed in L2 must be translated to L3. If L2 was chosen to be the same language as L3 (or a subset thereof), then this step is essentially vacuous (or close to it). If L3 was chosen to be a low-level language, then this translation is normally done by hand. It is possible to develop design aids to help in this translation. However, this is essentially the same as developing a compiler for L2, and, therefore, this option reduces to a previously considered option.

CONSISTENCY CHECKING

If L2 was chosen to be L3, consistency checking is inherited from the microcode design phase. If L2 and L3 are significantly different, the same activities performed in the microcode design phase must be duplicated here with respect to the new formalism, L3.

VERIFICATION

Again, if L2 was chosen to be L3, then verification is inherited from the microcode design phase; otherwise, the same kind of activity and design aids are appropriate here.

EVALUATION

This step is normally vacuous. If no new decomposition is performed and no more detailed information is now available, then the input performance constraints become the output performance constraints.

INFERENCE

This step is essentially vacuous.

INVOCATION

The microcode generation phase is invoked. Again, this may be done separately for each distinct module, a certain collection of modules, or the entire microcode design.

INTEGRATION

The activity in this step is identical to that in the microcode design phase.

PHASE NAME: Microcode Generation

PURPOSE

The purpose of the microcode generation phase is to produce the actual microcode required. This may involve translation and/or optimization.

INPUT

The microcode generation requirements include:

- Implementation specifications.
- Microinstructions specification.
- Hardware description.
- Performance constraints.

OUTPUT

The output of this phase is the microcode itself.

STEPS

FORMALISM SELECTION

This step is vacuous; L4 is given.

FORMALISM VALIDATION

This step is vacuous.

EXPLORATION

The specific translation tool or technique for transforming L3 into L4 must be selected. The implementor must decide whether local and/or global optimization is to be performed and, if so, what methods or tools are to be used. Optimization could also be done for either time or space.

ELABORATION

The elaboration step corresponds to translating the specification expressed in L3 into the microcode itself and optimizing that microcode. In the current state of the art, there is essentially always a tool for performing the translation (either a compiler or an assembler). Generic assemblers are available [ADVA78]. Generic

compilers are not yet common place although work has been done in this area [DASG78a, DASG80a].

Optimization can be done by hand or can be automated. Hand optimization may be best for short sequences of code; automated optimization is probably better for large segments of code. If automated optimization is appropriate, it is best to integrate the tool into an integrated package with the translator. There are many techniques for compacting the microcode [I. FR76, BARN78, DASG76, DASG78b, LAND80, MALL78, TOKO77, WOOD78, YAU74], and some have been integrated into HLMLs [PATT79].

CONSISTENCY CHECKING

This step corresponds to verifying that there are no compile/assembly errors. Such checking is normally incorporated into the translation tool.

VERIFICATION

If the optimization was done by hand, verification may be very time consuming due to human error. If the optimization was automated, verification can be done on the optimization tool itself (once) and the correctness of the microcode produced is inherited from the correctness of the tool.

Since the actual microcode is now available, testing (as described in the previous phase) can be performed on each separate microcode module.

EVALUATION

Now that the actual microcode is available, its performance characteristics can be evaluated and compared with the performance constraints developed in previous phases. If the actual hardware is available, these performance characteristics can be determined by executing on the hardware; otherwise, emulation (or simulation) can be used. These performance characteristics are made available to the previous phase.

INFERENCE

This step is essentially vacuous.

INVOCATION

This step is vacuous.

INTEGRATION

If each separate microcode module was passed to this phase, then this step is vacuous. If several modules were passed, they can be integrated together before making them available to the previous phase, or the integration can be done there.

REFERENCES

- [ADVA78] Advanced Micro Computers, "Advanced Microprogramming Development System -- System 29".
- [AGER76] Agerwala, T., "Microprogram Optimization: A Survey," IEEE Transactions on Computers C-25, 10, October 1976, pp. 962-973.
- [AH076] Aho, A. V. and Johnson, S. C., "Code Generation for Expressions with Common Subexpressions," Third ACM Symposium on Principles of Programming Languages, 1976, pp. 19-31.
- [ALLE70] Allen, F. E., "Control Flow Analysis," SIGPLAN Notices, Vol. 5, No. 7, July 1970, pp. 1-19.
- [ALLE76] Allen, F. E. and Cocke, J., "A Program Data Flow Analysis Procedure," CACM, Vol. 19, No. 3, March 1976, pp. 137-147.
- [BARN78] Barnes, G. E., "Comments on the Identification of Maximal Parallelism in Straight-line Microprograms," IEEE Transactions on Computers C-27, 3, March 1978, pp. 286-287.
- [BUSA69] Busam, Vincent A. and Englund, Donald E., "Optimization of Expressions in FORTRAN," CACM, December 1969, pp. 666-674.
- [COCK70] Cocke, J., "Global Common Subexpression Elimination," SIGPLAN Notices, Vol. 5, No. 7, July 1970, pp. 20-24.
- [DASG76] Dasgupta, S. and Tartar, J., "The Identification of Maximal Parallelism in Straight-line Microprograms," IEEE Transactions on Computers C-25, 10, October 1976, pp. 986-992.
- [DASG78a] Dasgupta, S., "Towards a Microprogramming Language Schema," Proceedings 11th Annual Workshop on Microprogramming (ACM), November 1978, pp. 144-153.
- [DASG78b] Dasgupta, S., "Comment on the Identification of Maximal Parallelism on Straight-line Microprograms," IEEE Transactions on Computers C-27, 3, March 1978, pp. 285-286.
- [DASG80a] Dasgupta, S., "Some Implications of Programming Methodology for Microprogramming Language Design," Proceedings IFIP TC-10 Conference on Microprogramming, Firmware and Restructurable Hardware, G. Chroust and J. Mulbacher, Eds., North-Holland, Amsterdam, 1980.
- [DASG80b] Dasgupta, S., "Some Aspects of High Level Microprogramming," ACM Computing Surveys, Vol. 12, No. 3, September 1980, pp. 295-323.

[DEWI76a] DeWitt, D. J., "A Machine-Independent Approach to the Production of Horizontal Microcode," Ph.D. Dissertation, University of Michigan, Ann Arbor, June 1976, Technical Report 76DT4, August 1976.

[DEWI76b] DeWitt, D. J., "Extensibility - A New Approach for Designing Machine-independent Microprogramming Languages," Proceedings 9th Annual Workshop on Microprogramming (ACM), September 1976, pp. 33-41.

[EARN72] Earnest, C. P., Balke, K. G. and Anderson, J., "Analysis of Graphs by Ordering of Nodes," JACM, Vol. 19, No. 1, January 1972, pp. 23-42.

[FRAI70] Frailey, D. J., "Expression Optimization Using Unary Complement Operators," SIGPLAN Notices, Vol. 5, No. 7, July 1970, pp. 67-85.

[GILL77] Gillett, W. D., "Iterative Global Flow Techniques for Detecting Program Anomalies," Ph.D. Dissertation, University of Illinois, Urbana, 1977, Technical Report UIUCDCS-R-77-848.

[GRAH75] Graham, S. L. and Wegman, M., "A Fast and Usually Linear Algorithm for Global Flow Analysis," Second ACM Symposium on Principles of Programming Languages, January 1975, pp. 22-34.

[HECH72] Hecht, M. S. and Ullman, J. D., "Flow Graph Reducibility," SIAM Journal of Computing, (1,2), June 1972, pp. 188-202.

[HECH73] Hecht, M. S. and Ullman, J. D., "Analysis of a Simple Algorithm for Global Data Flow Problems," SIGACT-SIGPLAN, October 1973, pp. 207-217.

[HECH74] Hecht, M. S. and Ullman, J. D., "Characterization of Reducible Flow Graphs," JACM, Vol. 21, No. 3, July 1974, pp. 367-375.

[HOPC72] Hopcroft, J. E. and Ullman, J. D., "An $n \log(n)$ Algorithm for Detecting Reducible Graphs," Proceedings Sixth Annual Princeton Conference on Information Sciences and Systems, 1972, pp. 119-122.

[KAM76] Kam, J. B. and Ullman, J., "Global Data Flow Analysis and Iterative Algorithms," JACM, Vol. 23, No. 1, January 1976, pp. 158-171.

[KENN71] Kennedy, K. W., "A Global Flow Analysis Algorithm," International Journal of Computer Mathematics, Sec. A, Vol. 3, 1971, pp. 5-15.

[KENN75] Kennedy, K. W., "Node Listing Applied to Data Flow Analysis," Second ACM Symposium on Principles of Programming Languages, January 1975, pp. 10-21.

[KILD73] Kildall, G. A., "A Unified Approach to Global Program Optimization," First ACM Symposium on Principles of Programming Languages, October 1973, pp. 194-206.

[LAND80] Landskov, D., Davidson, S., Schriver, B. and Mallett, P. W., "Local Microcode Compaction Techniques," ACM Computing Surveys, Vol. 12, No. 3, September 1980, pp. 261-296.

[MALL78] Mallett, P. W., "Methods of Compacting Microprograms," Ph.D. Dissertation, University of Southwestern Louisiana, Lafayette, December 1978.

[OSTE74] Osterweil, L. J. and Fosdick, L. D., "Data Flow Analysis as an Aid in Documentation, Assertion Generation, Validation, and Error Detection," Report No. CU-CS-055-74, University of Colorado, September 1974.

[PATT79] Patterson, D. A., "An Experiment in High Level Language Microprogramming and Verification," Unpublished manuscript, Department of Electrical Engineering and Computer Science, University of California, Berkeley, 1979.

[SCHA73] Schaefer, M., A Mathematical Theory of Global Program Optimization, Prentice-Hall, 1973.

[SMIT77] SMITE Training Manual, Report 30417-6002-RU-00, 1977.

[TOKO78] Tokoro, M., Takizuka, T., Tamura, E. and Yamaura, I., "A Technique of Global Optimization of Microprograms," Proceedings 11th Annual Workshop on Microprogramming (ACM), 1978, pp. 41-50.

[ULLM72] Ullman, J. D., "A Fast Algorithm for the Elimination of Common Subexpressions," Thirteenth Annual Symposium on Switching and Automata Theory, October 1972, pp. 161-176.

[WOOD79] Wood, W. G., "The Computer Aided Design of Microprograms," Ph.D. Dissertation, University of Edinburgh, Scotland, 1979.

[YAU74] Yau, S. S., Schowe, A. C. and Tsuchiya, M., "On Storage Optimization for Horizontal Microprograms," Proceedings 7th Annual Workshop on Microprogramming (ACM), 1974, pp. 98-106.

2.4 HARDWARE/SOFTWARE TRADE-OFFS

This topic was already introduced earlier in the context of the system design stage and receives additional in-depth coverage in Section 3 of this report which deals with the assessment of the family of TSD Methodologies. In this section we provide a brief review of the framework's perspective on this issue in its own right undiluted by all the other details involved in the presentation of the framework. The discussion starts with the definition of H/S trade-offs, outlines the approach prescribed by the framework, and identifies the main problem areas.

The problem embodied in H/S trade-offs is that of allocating the system's functionality between hardware and software components (be they off-the-shelf or custom designed) in a manner that satisfies all system design constraints. Because systems are perceived as H/S aggregates, the consideration of H/S trade-offs is perceived to be a central system design methodology issue. Its complexity is so high, however, that few methodologies make any attempt to deal with it, and most existing work focuses solely on computer systems selection, itself a difficult problem.

As far as the TSD Framework is concerned, the activities related to H/S trade-offs are distributed across the two phases of the system design stage: The system architecture design phase is engaged in a systematic process of reducing the binding options to the point where the binding phase is left to deal strictly with a selection among a few feasible alternatives. Every system architecture design decision, taken in the exploration step, has implications with respect to the type of technology that would be needed to realize the system. Furthermore, partitioning into hardware and software needs to be carried out as part of this phase because all performance models used in the evaluation and inference steps demand, as a minimum, information about the distribution of the system's functions among various processors and about interprocessor communication costs. All such design decisions are actually subject to explicit review and analysis in the inference step. Of particular concern for the inference step is to reject any design solutions which limit the range of feasible binding options unnecessarily. Since the system architecture is presumed to be developed top-down, the option elimination process is characterized by an iterative sequence of refinements and inferences.

Having the range of binding options significantly reduced by the previous phase, binding concentrates on selecting specific components among those still eligible. It is critical to proceed with the selection of individual components in the context of the entire system, and not by optimizing local decisions. This enables the focus to remain on the performance objectives of the system as a whole (cost included), where it belongs.

Neither option reduction nor component selection is a simple task. The former requires significant experience with system design and a good grasp of existing technology and current technological trends, issues that are difficult to formalize. The availability of appropriate performance models applicable both for performance evaluation and technological inferences could, however, assist the designer in very important ways. While the number of conceivable binding options may be overwhelming, the development of reduction strategies and performance models for a few common ones is believed to be feasible, but nontrivial. Similar challenges are present in dealing with the binding phase. On one hand, there is a need to develop adequate selection strategies for both software and hardware components. On the other hand, it is necessary to establish meaningful mappings between performance attributes present in the performance models mentioned above and those recognized in the actual component candidates.

The extent to which we are able to deal with some of these problems is treated explicitly in Section 3 of this report.

2.5 SYSTEM LIFE-CYCLE ISSUES AND THE TSD FRAMEWORK

2.5.1 DEVELOPMENT

In the normal system life-cycle, the "development" aspect refers to a sequence of logically equivalent systems descriptions. These begin with a high-level specification and continue through successively lower-level specification refinements until an implementation level is reached. Successive description levels are usually baselined to serve as milestones in the development process. For systems that involve only software, this approach is well documented [IBM80, JENS79]. However, when the system requires the possibility of a hardware/software tradeoff, then the methods that are designed to apply to software development begin to fail.

The TSD Framework provides an intellectual control that is the key to an orderly overall systems development process. In particular, the key concept of successive refinement is retained, but broaden in concept to include the H/S possibilities. Every phase of the TSD Framework accepts as an input the specifications of a set of system requirements. Each phase then transforms these requirements into a more detailed refinement by a fixed sequences of steps. Although the framework does not specify how the refinement is to be done, it does impose certain characteristics on any methodologies used for the task. The sequence of steps specified in each phase forces the methodologies to consider all issues pertinent to the given level of system development within that phase.

Because the process of moving from the TSD Framework to selected methodologies was discussed in the beginning of Section 2, there is no need for repetition here. The procedure is further illustrated in Section 3 where appropriate TSD methodologies for several application areas are introduced. The relations between the TSD Framework and the analysis, enhancement, and maintenance parts of the system life-cycle will be discussed next.

REFERENCES

- [IBM80] IBM System Journal 19, No. 4, Special Issue On Software Development, 1980.
- [JENS79] Jensen, R. W. and Tonies, C. C. Software Engineering. Prentice Hall, 1979.

2.5.2 ANALYSIS

System analysis is the process of assessing some performance property of the system by examining it or its specifications. In the first case, it involves monitoring the actual system behavior and evaluating the collected data with respect to the property of interest. Alternately, one uses the specifications to construct a model of the system being investigated. Thus, the model replaces the system as the object of the analysis and, consequently, the conclusions being drawn are valid only to the extent to which the model may be shown to be faithful to the real (existing or postulated) system. In either case, both quantitative and qualitative aspects of the system may be subject to analysis.

The TSD Framework indicates that analysis is part of each phase and is concentrated mainly in the evaluation, inference, and integration steps. During evaluation, analysis is based on system models derived from specifications available in that phase and is aimed at supporting both development and enhancement activities. Furthermore, it is the fundamental mechanism through which performance parameters (constraints) are assigned to newly identified lower level components in a manner which assures that the constraints acting upon the upper level components are guaranteed to be met if the lower level ones are satisfied. Besides assisting in the propagation of constraints, analysis is also instrumental in rejecting any design solutions that are clearly unable to meet stated constraints.

The models and the data generated by the evaluation step are the basis on which the analysis process draws various conclusions in the inference step. They deal with all facets of performance including feasibility, forecasting, technological consequences, environmental impact, etc. Again, both development and enhancement take advantage of this instance of the analysis in similar ways. Negative results are used to accept or reject proposed design solutions or enhancements.

In contrast with the other two steps, integration has available to it the actual system or parts thereof. As a result, the starting point for the analysis is monitoring system behavior under various benchmarks (actual or synthetic). The goal is to determine if all assumptions made earlier are satisfied and all relevant constraints are actually met (even though the assumptions are proven correct the models relating them to the active constraints could be shown to be invalid). Furthermore, whenever this is not the case the analysis is directed toward identifying the source of the discovered problem. Development, enhancement, and maintenance involve this aspect of analysis in analogous manner.

Because analysis is rarely done for its own sake, it was to be expected that its goals vary with those of the activity to which it is subordinated. Nevertheless, development, enhancement, and maintenance appear to employ analysis in a similar manner while placing different emphasis on one step or another. Maintenance, for instance, deals primarily, but not exclusively, with the type of activities present in the integration step. It is reasonable, therefore, to conclude that the framework's ability to characterize methodologies does not exclude analysis and should prove to be an invaluable aid in better understanding

its relationship with the other aspects of the system life-cycle.

2.5.3 ENHANCEMENT

It is not unusual for the needs of an application to gradually change with time. As a result, even if a given computer based system initially fulfills all the needs of an application, the suitability of the system diminishes with time. One solution to this problem is to replace the system when its capabilities become too limiting. Another solution is to periodically modify the system so as to enhance its capabilities. The choice of one approach over the other is primarily a matter of economics and involves considerations such as the cost of lost productivity due to limited system capabilities, the current investment in equipment and software, and the costs involved in making system modifications. The first two considerations strongly favor enhancement, but the latter may rule it out if the system has not been designed in a manner conducive to the making of modifications. For this reason, a fundamental objective of the development process should be the design of systems that are easy to modify.

From the framework viewpoint, the process of enhancing a system is the same as the process of developing a system. There must be a problem definition stage to identify the current needs of the application domain and to identify system enhancements that meet these needs, there must be a system design stage in which the system-level revisions are designed, and there must be a hardware design stage and a software design stage in which the hardware and software revisions are designed. The distinguishing aspect of enhancement is that all deliberations in these stages are constrained by the need to be compatible with the existing system. The effect of this constraint is to severely limit the range of options that are feasible.

Economic considerations play an important role in any design effort, but are more significant in enhancement than in development. The reason is that during development, the cost of doing a system right (creating a modular, easy to understand structure) may cost no more than doing it otherwise. In the case of enhancement, revisions may not fit naturally into the existing structure, and the cost of revising the structure to cleanly incorporate the changes may cost much more than a "quick" fix. However, quick fixes make the system structure more complex and this makes subsequent modifications more costly. The issues are quite complex, especially for large systems, and a good discussion of this is given in [BELA79]. In particular, it is quite possible that the best strategy, from the viewpoint of total life-cycle costs, may be to employ sequences of quick fixes followed by periodic restructuring efforts. Because quick fixes are contrary to the general rules of good design practice, it is clear that the system requirements associated with enhancement efforts must include specific instructions regarding this issue.

The task of enhancement is greatly facilitated if the original system was designed under a TSD methodology, if the design documents are accessible from a local database, and if there are software tools for performing various analysis tasks that are necessary to the design

process. The benefits of this are several. Because the original design was done under a TSD methodology, the design decisions are recorded and the sources of constraints are traceable. This information allows the designer to evaluate the effect of proposed changes on the entire system behavior and thus reduce the risk of side-effects. Having the design information in a local database and having software tools to aid in analysis reduces the time and effort needed to identify acceptable design changes. Finally, if the redesign is done under a TSD methodology and the database documentation is appropriately updated, the maintainability and the enhanceability of the system will be preserved.

REFERENCES

[BELA79] Belady, L. A. and Lehman, M. M., "The Characteristics of Large Systems," in Research Directions In Software Technology, MIT Press, pp. 106-138, 1979.

2.5.4 MAINTENANCE

The continuing growth in required complexity makes it unrealistic to test any useful system exhaustively as part of its development. Granting even the most enthusiastic attention to systematic evaluation, it simply is impractical to route a system through each possible decision path that it could follow. As a result, there is a likelihood that, at some point during its routine use, a system's decision mechanism will select a previously untested course that happens to produce a malfunction. This behavior, affecting both hardware and software, is distinct from the wear and tear traditionally associated with physical equipment. When these factors are considered together, it is clear that maintenance is an unavoidable aspect of any system with which we are to be concerned. Treatment of this situation as a reality rather than an outbreak of pessimism makes it compulsory to consider the need for maintenance as a fundamental system issue. In fact, maintainability represents a property to be treated as an integral part of the concerns from the start of the system development process.

Acceptance of maintenance as an inevitable requirement exerts an influence throughout the major development stages. Awareness of this need at the problem definition stage, for example, establishes the impetus to include maintainability as one of the basic system requirements. This serves a useful purpose even before the system is bound to a distinct architecture because it allows the designers to focus on the weak points inherent in the application, independent on any particular design. Once these potential trouble areas are identified, the need to address them can be included among the system requirements generated by this stage. It is not at all surprising to see these considerations manifest themselves as serious constraints on those requirements.

Once the dominant activities move to the system design stage, maintenance considerations expand to include those related to a particular configuration. It is at this point that the designers can begin to assess the effects of maintainability requirements on H/S tradeoffs. For instance, a seemingly attractive hardware solution may be less so (in comparison to a software approach) when one includes the relative burdens imposed on each alternative by maintainability requirements. The resulting set of eligible entries to the binding phase would be tempered accordingly.

The resulting requirements that are made available to the hardware and software design stages reflect the ongoing concern with maintenance. On the hardware side, this means that error detection, fault tolerance, and component modularity are prominent factors influencing the selection of off-the-shelf equipment and the specifications for custom hardware. The resulting requirements propagate to each phase of the component design stage, ultimately producing circuits and electromechanical components in which maintainability is an inseparable aspect. Analogous concerns are included as part of the software and firmware design efforts. Thus, concepts such as modularity and simplicity of interfaces between modules are not viewed exclusively as vehicles for simplifying development. Rather, they can also be exploited to facilitate the process of localizing software errors and correcting them with minimum impact on the rest of the

system. (The actual processes associated with error correction are conceptually identical to those involved in enhancement and, therefore, are not addressed here.) Similarly, emphasis on error reporting is not limited to hardware design. Using information developed during previous stages (including that relating to the application's intrinsic weaknesses), the program design requirements can be defined to include features whose specific purpose is to reveal certain aspects of the software's behavior so that potential trouble spots can be monitored and malfunctions can be detected quickly.

Since maintainability parallels other system concerns throughout this succession of stages, the resulting documentation will include helpful information about the nature and use of the system's maintenance-related features. Accordingly, the system's users will be in a position to take full advantage of these facilities when such needs arise. Additional help can be obtained from development facilities. Besides their primary use, such vehicles offer excellent opportunities to identify implementation-dependent weak spots and other potential trouble areas that were not identified in other ways.

Because of its emphasis on the H/S dualism, the TSD Framework accommodates sustained attention to maintenance quite comfortably. Since maintainability is a basic property that transcends the particular implementation selected to meet a given set of requirements, its characteristics can be defined abstractly for the system being considered. Then, when the system design is bound to a specific H/S configuration, maintainability is included among the objectives addressed by that design. For instance, the designer can determine which aspects of system performance to monitor prior to any specific configurational commitment. Exactly how the hardware and software will be instrumented to report on these aspects is an issue to be addressed in subsequent stages. Thus, concerns for maintenance are attended to along with the others at each stage. Consistent with the TSD Framework's intent, such attention can be assured regardless of the way the maintainability requirements are perceived or the method used to decide how the requirements will be met.

2.6 CONCLUSIONS

The development of the TSD Framework goes beyond the mere consolidation of current understanding of system design approaches and methods under an unified umbrella. It demonstrates, first of all, the potential benefits derivable from the use of the methodological framework concept in studies concerned with methodology characterization, evaluation and development. Its ability to focus strictly on the basic decision mechanisms involved in particular methodologies simplifies the investigation, aids in the discovery of possible omissions, and directs the researcher's attention on the main methodological objectives, be they explicit or implicit. Thus both better understanding and easier redefinition of the goals are enabled. By providing for the readjustment of the objectives prior to restructuring the methodology, a new and more rational approach to methodology development is made available.

The TSD Framework also promises to place on a more rigorous basis the notions of phase and step. The former is defined based on the identification of the knowledge domain that appears to support its activities. It is also shown that the sharing of similar ultimate objectives among phases results in a unified phase structure that consists of steps that are either fundamental to all design endeavors or supportive of one of the common objectives such as early error detection or systematic performance of H/S trade-offs analysis.

Furthermore, as part of the TSD Framework consolidation some contribution is made toward establishing a novel perspective on system integration. A rigorous scrutiny of its role in system design shows it to be an essential part (i.e., step) of each phase and not a phase in its own right.

Finally, the work being reported here demonstrates that a systematic H/S trade-offs strategy must acknowledge the presence of these trade-offs as an important issue in all phases of the framework (as part of a technological 'inference' step). The reason for this being the fact that all design decisions affect the range of available choice for both software and hardware binding. Moreover, trade-offs analysis similar to that employed for software and hardware is also present in phases that deal solely with software or hardware design.

In conclusion, one could state that the results obtained so far strongly support the conjecture that the TSD Framework has the potential to play a significant role in future methodological advances. The stage is set now for instantiating the framework into several system design methodologies aimed at supporting effective design in key application areas.

3. ASSESSING THE FAMILY OF TSD METHODOLOGIES

3.1 INTRODUCTION

The goal of this section is to assess the family of TSD Methodologies with respect to its ability to effectively meet the system design objectives of several common DoD applications: embedded systems, information processing systems, and command, control and communication systems. The assessment consists of a feasibility study which illustrates the key features of the TSD Methodologies. It shows the way in which these methodologies approach system design and the techniques and tools that would make viable their transfer from the current research and development state into productive use.

The difficulty of such an undertaking hardly needs to be argued. Methodologies, in contrast with the framework they instantiate, are problem and environment dependent. They owe their effectiveness largely to the extent to which they are able to take advantage of the characteristics of the application through the use of appropriate techniques. Furthermore, the usefulness of a methodology also depends upon a correct match between the techniques it employs and the environment in which it functions, i.e., the organization, the people, the available technology and expertise, etc. Consequently, consideration of the entire range of systems being grouped under the three generic categories introduced earlier is deemed impossible in view of the great variety of applications encountered in the defense field.

The information processing systems category, for instance, includes both cartographic databases such as those seen at the Defense Mapping Agency (DMA) and logistics command databases; they are, however, quite distinct in nature. Similar heterogeneity may be observed in the other two groups. Moreover, even when two applications seem to have many features in common, they may be subject to different sets of design constraints which lead to methodological variations. Such is the case, for example, when one compares functionally similar embedded systems present in a manned versus unmanned spacecraft.

As a direct consequence of these facts and other considerations explained below, several limitations have been imposed over the scope of the TSD assessment.

- Three classes of systems are considered, one for each of the three application areas above. Furthermore, each class is characterized by certain so-called 'characteristic' features thus hiding some of the variability between systems supporting similar application domains.
- This general view of the application areas is coupled with an equally high level treatment of the corresponding methodologies. Consequently, the methodologies being outlined in the study would necessitate further refinement if the problem domain and the environment in which they are to be employed are reconsidered at a greater degree of specificity.

- Another scope limitation is the result of the fact that not all areas covered by the TSD Framework are of equal significance for this study. Software and hardware design, for instance, receive minor coverage because these design activities are relatively well understood and because a lot has been written about them already. (See Section 2 for an overview of existing technology and appropriate references.) Special attention, however, is given to system-level design issues, in general, and to the complex issue of hardware/software trade-offs, in particular.
- The feasibility of the TSD Framework for command, control and communication systems is demonstrated only by implication; since such systems are understood to be a composite of embedded and information processing systems, the benefits the last two groups derive from the TSD technology are also enjoyed by their composite. Section 3.2.3 considers this point of view in more detail and shows the extent to which it represents a useful working hypothesis as well as the design complexities it ignores. In that section it is also explained that a separate and more detailed study of a TSD Methodology for command, control and communication systems is considered unwarranted at the present since a better understanding of methodologies for the design of embedded and information processing systems is a prerequisite for a more in-depth investigation of command, control and communication systems.

The TSD assessment starts with an examination of the essential characteristics of the embedded, information processing, and command, control and communication systems (Section 3.2). The unique nature of the applications supported by DMA is used to emphasize the dependency between methodologies and the nature of the organization that may employ them (Section 3.3). The point is made that future detailed assessments of the TSD technology ought to be carried out not only with respect to a specific class of systems but also with respect to the type of organization that intends to build them.

In Section 3.4 a class of TSD Methodologies whose scope is limited to the system design stage is introduced. By relegating the formal characterization of the class to Appendix E, the presentation is kept informal. The emphasis is placed on the design strategy featured by the TSD Methodologies. The feasibility of the approach and its ability to adapt to a large variety of systems (of the embedded and data processing type) is demonstrated in Sections 3.5 and 3.6. The principal results of the assessment are reviewed below.

- By accomplishing the transition from the TSD Framework to a class of distributed system design methodologies and by describing how one could employ these methodologies on system design projects having characteristics common to a multitude of DoD (including DMA) type systems, the technical feasibility of the TSD Framework is demonstrated.

- The actual use of the concepts and methodological study approaches developed during the consolidation effort described in Section 2 (in particular the synthesis of methodologies given a framework and a class of applications) illustrates convincingly the assistance these approaches could provide to methodological research and development.
- The TSD Methodologies are shown to promote a systematic approach to the performance of hardware/software trade-offs thus avoiding the known problem of premature hardware procurement. Future research advances in this area combined with experiments in which these methodologies are applied to real-life systems hold the key to making the employment of these methodologies both practical and profitable in terms of quality and productivity gains.
- Techniques and tools (available or postulated) identified as necessary for productive use of the TSD Methodologies form the starting point for the development of the TSD Facility master plan introduced in Section 4. It must be noted, however, that, as indicated in Section 4, there are many other factors that intervene and influence the planning of such a facility in addition to the techniques suggested by the use of one methodology or another.
- Four by-products of this study are:
 - a methodology definition language (Appendix D) which holds the promise to reduce some of the ambiguities currently found in most of the methodology literature and which may be useful as a methodology enforcement and project planning tool;
 - a formal characterization of the nature of the specification languages involved in system design and of some of the criteria associated with the verification of the proposed designs (Appendix E);
 - an investigation in formal approaches to system requirements definition (Appendix F);
 - a proposal for a distributed system design specification language (Appendix G).

3.2 CHARACTERIZATION OF THREE CLASSES OF DOD SYSTEMS

Any characterization of the extreme diversity of systems encountered in the defense field is bound to produce some controversy. This is, in part, due to the fact that systems rarely fit cleanly in the niches created by one taxonomy or another. For this reason, this section describes not a taxonomy of DoD systems but a set of working hypotheses whose role is to establish a precise context for the discussions to follow. Rather than trying to partition systems into distinct categories (i.e., equivalence classes), the approach adopted here is to identify several types of systems which, as a group, are able to cover the important characteristics of the systems in existence today. In other words, the nature of an actual system should be describable in terms of a composition of two or more idealized classes of systems to which it belongs.

Three such classes are recognized here: (1) embedded systems, (2) information processing systems, and (3) command, control and communication systems. While the existence of these classes has long been acknowledged and the terms are common in the literature, their meaning differs from one report to another. This report attempts to associate with each class those characteristics that seem to be commonly recognized by most authors. All systems that do not match exactly the definitions of any of the three classes are assumed to be made of subsystems which fall cleanly in one of these classes.

Even under these simplifying assumptions, the difficulties associated with the development of system design methodologies for each of the three system classes are not completely overcome. The extent to which some methodology is applicable to all systems of a given type is still a major concern. Most problems stem from the great variability in the design constraints associated with each system being developed. Since the distinctions are not merely quantitative but also qualitative in nature, design methodologies may differ significantly, if not in the overall strategy, at least with respect to the specific design techniques being employed.

The reliability requirements of a system embedded in a communication satellite, for instance, are significantly more stringent than those placed on an air traffic control system (particularly when considering them in conjunction with other active constraints such as possible maintenance procedures, weight, size, power, shielding, etc.) and they result in the employment of drastically different system architectures. A methodology aimed at the entire class is unable to recognize such fine differences between the two instances of embedded systems. Nevertheless, proper design of the methodology ought to enable further refinement of the methodology so as to take advantage of the particular combination of constraints. Otherwise the methodology may prove unfeasible for many systems in the given class. While the characterizations that follow and the methodologies proposed in later sections have been carefully selected so as to avoid this pitfall, only through the actual use of the methodologies one is able to provide the ultimate validation of having achieved this goal.

3.2.1 EMBEDDED SYSTEMS

While it is true that, from some point of view, all computer systems are actually 'embedded' in some larger system such as a logistic command center, a vehicle, a weapon, a communication network, etc., the term embedded system is used to identify a class of systems having the following main characteristics:

- They interact in real-time with electronic devices such as sensors, radars, etc. by receiving inputs and/or by controlling the activities of such devices.
- They are locally distributed, if at all.
- They provide a service critical to the system in which they are embedded and require extremely high reliability. In other words, their function is essential for the operation or survival of the larger system they support. The on-board computers of both manned and unmanned aircraft are generally relied upon to assist at all times in the navigation procedures and their failure may lead even to the loss of the craft. Similarly, a computer failure on a communication satellite may hamper the normal activities of an entire organization.
- They are often required to perform their functions in rather restrictive environments such as on board ships, in outer space, etc. and may be subjected to severe weight, power, and volume limitations as well as to electromagnetic interference, radiation, etc.
- Their human interfaces, when present, demand elaborate human engineering.
- They need small databases but they may be involved in the collection of large volumes of data for later processing or for purpose of supplying it to some other system for analysis.
- Their evolution is determined by external changes in the goals of the systems they support (e.g., a changes in the mission to be performed by some military system).
- Their security against unauthorized access is achieved by measures that secure the larger system in which they are embedded. Therefore, security considerations do not affect significantly the system design.

3.2.2 INFORMATION PROCESSING SYSTEMS

With respect to information processing systems, there is greater agreement on their definition:

- They consist of large databases and access mechanisms for updating and retrieving the information present in the databases.
- They are often geographicly distributed in order to improve data accessibility and availability for the various components of the organization being supported.
- They are rarely subject to meeting real-time processing constraints. However, they are often required to maintain data generated or used by real-time devices such as graphic displays.
- They interface with humans (to a growing extent) via interactive terminals which are required to meet certain response time constraints. The human factors, while somewhat less critical then in the case of embedded systems, are still very important in making the system an effective tool for the organization.
- Their throughput is viewed as a key parameter measuring the volume of work they are able to carry out.
- Their security is a major concern particularly when the data they control is of a sensitive nature. Furthermore, measures that secure the physical location of the system are insufficient and complex mechanisms need to be built into the system in order to prevent unauthorized access to its data.
- They are generally built with off-the-shelf components but they may include small highly specialized custom-made devices. In the future, however, the role of custom-made components may increase in importance thus making room for new hardware/software alternatives to be considered.
- Their reliability is important, but they tend to be more tolerant toward faults because of the ease with which human intervention may take place. Furthermore, since information systems are not subject to the same extreme physical constraints placed over embedded systems, more resources are available for use in the error detection and recovery.

3.2.3 COMMAND, CONTROL, AND COMMUNICATION SYSTEMS

In the simplest way, command, control and communication (C3) systems are mere assemblages of embedded and information processing systems. A ballistic missile defense system, for instance, could be treated as being composed of several cooperating subsystems. Among them, those that support the individual radar posts and the weapon dispatching and guidance fall in the category of embedded systems. The remaining subsystems are of the information processing type. They are repositories of information received from other subsystems and of tools that use this information to support the decision processes for which the respective command centers are responsible.

This view of C3 systems, while correct, is incomplete. It is acceptable as long as one is concerned only with showing that the design of C3 systems presupposes the availability of design methodologies for embedded and information processing systems. The aspects not being captured by this view, however, are the additional functional and performance constraints placed on the component subsystems by the very nature of the C3 system and the role played by communication. Some of these issues are elaborated below.

- The communication between subsystems and between a subsystem and the devices with which it interacts becomes the most critical aspect of system design. A battlefield information distribution system, for example, interfaces with troops operating on enemy territory, with command posts, with intelligence gathering devices, weapons, etc. The problems related to assuring reliable communication, security, and continued operation in the presence of communication, device, and/or subsystem failures and potential subversion are complex.
- All the constraints recognized in the design of information processing systems are present and exacerbated in the subsystems of a C3 system. Both throughput and response time have to meet sudden load increases placed on the system by critical and fast evolving situations such as an enemy attack. Moreover, the vital role played by the system, combined with increased hostility in the environment in which it functions, demands stricter security measures.
- The assumptions made about the data and the way it is being used also differ from an information processing system. Potential failures in reporting, intelligence, and communication may render data to be either inaccurate or incomplete. For example, a temporary communication cutoff between two command posts may necessitate decisions to be made based on estimates of what might be happening at the other post. Furthermore, because the primary function of C3 systems is to support the decision making process (tactical, logistic, etc.) easy development of appropriate models for evaluating the potential consequences of alternate strategies must exist in addition to the ability to query the databases.

In view of these facts, a methodology for the design of C3 systems appears to involve several, somewhat independent, subtasks.

- The first one is to find acceptable solutions to maintaining reliable communication and to adopt an overall policy with regard to how one is to deal (from a military point of view) with communication failures. The former issue falls more on the shoulder of the communication technology while the latter involves strategic considerations. Neither of the two is within the scope of this research.
- The second subtask is to separate the C3 system into its component subsystems of the embedded and information processing types. The separation is based upon the intrinsic distribution of functions between the entities forming the larger system being supported, upon strategic and other military considerations, and upon the communication technology being employed.
- The next subtask is to define the constraints being placed upon each of the subsystems.
- Having established both the functional definition and the relevant constraints of each subsystem, its design may proceed in a manner appropriate for the respective class of systems to which it belongs.

The peculiar relationship between the three classes of systems characterized in this section makes a separate assessment of the TSD Methodologies with respect to C3 systems unnecessary. The development of a TSD Methodology for C3 systems presupposes the availability of design methodologies for the other two types of systems thus clearly identifying both the feasibility and the importance of the TSD technology in the area of C3 systems. Consequently, the decision has been made to focus the more detailed investigation on the design strategy promulgated by the TSD Methodologies and to illustrate it only for embedded and for information processing systems. The results are presented in Sections 3.4 and 3.5.

3.3 SYSTEM DESIGN NEEDS AT DMA

The Defense Mapping Agency (DMA), like most other DoD organizations, depends extensively on the support of computer based systems in order to fulfill its role in the DoD community. Being responsible for satisfying the mapping, charting and geodesy (MC&G) needs of all the military organizations presents, from a system design perspective, some advantages and many challenges. On one hand, a key advantage could be the fact that by concentrating on a narrower set of applications one increases the chances for fast meaningful progress in the establishment of effective methodologies and facilities for use in system design. On the other hand, however, there are two major difficulties that need to be overcome: (1) the production pressures which leave few resources to be dedicated to the means of production and (2) the unique and complex nature of DMA systems involving numerous and extremely large cartographic databases as well as specialized devices used to process some of the data.

In order to understand the methodological needs of the DMA, one has to consider the characteristics of the production environment existing at DMA and the nature of the applications with which this organization is involved. In this regard, the following issues seem to have the greatest bearing on the future of system design at DMA.

- The DMA production plan is determined by the MC&G defense needs of the many DoD organizations. Changes in the data format, use and collection (quite often unanticipated) bring about increased demands for MC&G products, demands that translate into corresponding enhancements in the systems employed by DMA. Its ability to keep up with future growth indicates a need to employ effective system design methodologies capable of supporting the dynamic evolution experienced by DMA systems.
- While at present most DMA systems could be considered to be of the information processing type, their MC&G nature makes the importation of system design technology somewhat less direct. For an extensive discussion of the basic distinctions between business and geographic data processing the reader is directed to [NAGY79] which also contains a survey of the major geographic data processing systems in production today (including those operating at DMA). The following is a list of features identified in [NAGY79] as being unique to geographic data processing:
 - demanding performance constraints not present in other data processing applications;
 - presence of locational attributes;
 - two-dimensional nature of the problem domain;
 - particularly large amount of storage;
 - lack of commercially available systems;
 - government ownership of most existing systems;
 - specialized and expensive input/output devices;
 - dependence upon remote sensing technology.
- All major geographic data processing systems in production today

have been developed by some government organization (within or outside the U.S.A.) and have been designed to serve a set of very specific requirements. Consequently, geographic data processing for military purposes receives little attention outside the Government and puts DMA in the position of having to develop on its own the system design technology required to maintain and enhance its MC&G production.

- The complexity of the current types of systems is on the rise. The number and volumes of the databases, the workload, and the number and variety of products all experience noticeable growth. Moreover, greater interdependencies between databases and products is anticipated. The ultimate consequence of these trends might be the evolution of a single distributed DMA system, a critical component of the entire organization.
- There is also evidence pointing to a possible new group of systems of the embedded type. Computer controlled devices in use at DMA can be viewed to be in this category already. Furthermore, any increased future involvement of the organization in the data collection process most certainly is bound to extend DMA related system design efforts into the embedded systems area.
- At a more speculative level, incorporation of DMA systems into larger C3 systems can not be ruled out. Major increases in the data collection rate combined with a need to possess extremely current MC&G products (possibly on-line) may contribute to making this qualitative jump.

The productivity associated with the generation of MC&G products at DMA appears to be related to the quality of the computer based systems being employed, which in turn depends on the effective use of current technology at hardware, software, and system levels. TSD Methodologies hold the potential to assist DMA with many of these system related problems and to provide cohesiveness to long range planning in this area. They extend the ability of the organization to control and manage system development, maintenance, and enhancement. Furthermore, TSD Methodologies promote careful definition of system requirements and more effective use of available technology. In other words, the DMA's strides toward quality, productivity, enhanceability, maintainability, and low system design costs are identical to the basic objectives of the TSD technology.

Although the general orientation of this assessment is not DMA specific, the impact of the TSD Methodologies on DMA related system design efforts is apparent and the use of DMA inspired case studies only enforces it further. Future advances in this direction, however, require some fine tuning of these methodologies and additional experimental work on real DMA systems. Moreover, the TSD Facility master plan presented in Section 4 relates directly to and is consistent with current efforts aimed at the establishment of a DMA modern programming environment (MPE). The MPE work is leading to the establishment of a TSD Facility at DMA, a facility whose scope is limited to software development. While current MPE efforts focus on the selection of specialized tools, future work will have to emphasize

the integration of these tools. The TSD Facility description appearing in Section 4 includes the requirements definition for the integration process.

REFERENCES

[NAGY79] Nagy, G. and Wagle, S., "Geographic Data Processing," Comp. Surv. 11, No. 2, pp. 139-181, June 1979.

3.4 TSD VIEW OF DISTRIBUTED SYSTEMS DESIGN

3.4.1 INTRODUCTION

The system design stage covers all design activities involved in taking a set of system design requirements and generating the specification of the hardware and software requirements for the respective system. There are two phases that make up this stage: the system architecture design phase and the system binding phase. The former deals with the selection of an overall system architecture which accomplishes the intended system functionality and which, under a reasonable set of technological assumptions, meets the performance and other constraints originating with the system requirements. The proposed architecture and all the design decisions taken during this phase form a processing model used as input to the binding phase.

The binding phase, based on the limited degrees of freedom still left open by the system architecture design phase and based on market availability, identifies a particular mix of software and hardware and produces specifications for all needed components. The nature of the specifications, however, may vary from component to component depending on its intended realization (software or hardware) and on the manner in which it is to be obtained (off-the-shelf, through customization, or custom-made). The system design stage is also concerned with the integration of the system components from the point when both the software and the hardware components are available and up to the point when the system is offered for customer acceptance testing.

A system design methodology, like all other design methodologies, has three facets: one or more specification languages, a design strategy, and an appropriate set of design/analysis techniques. Because Section 3.4 is concerned with identifying not a specific methodology but a class of TSD Methodologies, these three issues do not enjoy equal treatment.

- SPECIFICATION LANGUAGES. The system requirements, the processing model, and the hardware/software requirements define the specification language needs of the TSD Methodologies. Sections 3.4.2 through 3.4.4 offer informal definitions of the general nature of these three types of specifications. Formal definitions are included, however, in Appendix E. It establishes the theoretical foundation for the entire Section 3 and presents the interested reader with formal requirements definitions for the specification languages needed to support distributed system design. (The approach is similar to that used in [ALF079].) One may use the contents of Appendix E in both the design and the evaluation of certain classes of specification languages. While the design of particular specification languages is outside the scope of this investigation, an attempt has been made to illustrate potential directions that could be followed by future research efforts in this area. Consequently, Appendices F and G describe language proposals for system requirements definition and parts of the processing model.

- DESIGN STRATEGY. The design strategy is first introduced in Section 3.4.5 in the form of a tutorial. The strategy is later formalized in Section 3.4.6 by using the methodology definition approach described in Appendix D. Formal definition of the relationship between the design strategy and the nature of the specification languages involved is relegated to Appendix E.
- TECHNIQUES. In the absence of particular specification languages, the techniques are only touched upon. Their objectives are suggested by the strategy and by the nature of the specifications, but no specific techniques are proposed here.

REFERENCES

[ALFO79] Alford, M., "Requirements for Distributed Data Processing Design," Proc. 1'st Int. Conf. on Distributed Computing Systems, pp. 1-14, October 1979.

3.4.2 INFORMAL DEFINITION OF SYSTEM REQUIREMENTS

The system requirements are generated in the problem definition stage. They consist of a conceptual model and a set of constraints which together define the acceptability criterion for any proposed system realization: a system is said to meet its requirements if and only if it carries out the functionality described by the conceptual model and satisfies all the constraints present in the system requirements. (Note, however, that implicit in this definition is the existence of a non-empty, usually infinite, set of systems that are able to carry out the desired functionality and an effective procedure by which to determine if a given system does or does not satisfy all the constraints.)

The role of the conceptual model is to capture in finite and precise terms the nature of the interaction between the needed system and its environment. In general, the conceptual model must have the ability to describe the relevant environmental states, an abstraction of the states of the system, and the way in which both the environmental and system states change. The approach to describing the states and the state transition rules varies from one specification language to another. The language discussed in Appendix F, for instance, employs a set-theoretical notation to describe both the environmental and the system states and uses predicate calculus to define the state transition rules. By contrast, other languages promote operational approaches based on data flow graphs [BELL77], applicative methods [ZAVE81], etc.

Furthermore, some languages make implicit assumptions about either or both the nature of the states and of the state transition rules; the loss in generality is motivated by increased specificity in the handling of a particular application area. As an example, a system that responds to stimuli from the environment in a manner which is independent of the history of previous stimuli and responses may be easily described in a

language which equates the state of the environment with the current stimulus, has no ability to describe system states, and is capable of defining a mapping from the set of stimuli to the set of responses. Yet another example could be used to illustrate the fact that there is also great variability in the way state transitions may be described: in a biomedical simulation system a new state is generated as a result of the integration of a set of differential equations.

Increases in the ability to formally define the desired functionality are not accompanied by commensurable advances in the definition of system constraints. There are four important reasons contributing to this. First, there is a great diversity of types of constraints (e.g., response time, space, reliability, cost, schedule, weight, power, etc.). Second, some of them are related to possible design solutions which are not yet formally stated at the time the system requirements are being conceived. Furthermore, their relevance differs at different points in the design. Third, many constraints (e.g., maintainability) are not formalizable given current state-of-the-art. Finally, not all constraints are explicit. For instance, the designer is expected to follow generally accepted rules of the trade in designing a system without having them explicitly stated.

REFERENCES

- [BELL77] Bell, T. E., Bixler, D. C. and Dyer, M. E., "An Extendable Approach to Computer-Aided Software Requirements Engineering," IEEE Trans. on Soft. Eng. SE-3, No. 1, pp. 49-60, January 1977.
- [ZAVE81] Zave, P. and Yeh, R. T., "Executable Requirements for Embedded Systems," Proc. 5'th Int. Conf. on Soft. Eng., pp. 295-304, March 1981.

3.4.3 INFORMAL DEFINITION OF PROCESSING MODEL

The methodology put forth in Section 3.4 treats systems as being describable by a hierarchy of related design specifications where the specification at one level reveals a design solution for some problem which is formally defined within the level above. The processing model reflects this view by assuming a similar structure: a total order over a finite set of design specifications. The total ordering is not really necessary but has been adopted in order to simplify the presentation of both the processing model and the system design strategy. Furthermore, the extrapolation to an upside-down tree (a tree in which the level number of each node is defined as the longest distance from a leaf rather than root) is trivial.

By definition, each design specification is viewed as corresponding to a subsystem in the overall system. The support relation between subsystems is explained in the Section 3.4.5 and is formally defined in Appendix E. The remainder of this section focuses on the informal definition of the design specifications.

Regardless of its position in the hierarchy, each design specification consists of same six components:

- PROCESS STRUCTURE
 - network topology in terms of processes and links
 - definition of system and external processes
 - definition of links
 - definition of link communication protocols
- PROCESSOR STRUCTURE
 - network topology in terms of processors and interconnections
 - definition of processors
 - definition of processor interconnections
 - definition of interconnection communication protocols
- PROCESS/PROCESSOR ALLOCATION
 - allocation and reallocation rule
- PERFORMANCE SPECIFICATIONS
 - performance requirements of processes and links
 - performance requirements of processors and interconnections
 - performance characteristics of processes and links
 - performance characteristics of processors and interconnections
 - performance models
- BINDING OPTIONS
 - set of feasible realizations of the process and processor structures
 - set of binding constraints
- CONSTRAINTS.

The PROCESS STRUCTURE describes the subsystem functionality by means of a network of communicating processes interconnected via links. Each link provides a logical connection between two or more processes. The message traffic on each link, however, behaves in accordance with a communication protocol specified by the designer. In the top level subsystem the processes may correspond to successive transformations of the input data in a data processing system or to query processing in a database system. At other levels the process structure may be describing operating system capabilities. In all cases, however, the description is independent of the way in which the processes are distributed within a realization of the system and of the manner in which they may be implemented.

The PROCESSOR STRUCTURE, in conjunction with the process/processor allocation explained below, is an abstraction of all the subsequent levels in the hierarchy. In its simplest form, the distinction between the process and the processor structures is like the distinction between an application program and the operating-system/hardware combination that enables it to execute. Furthermore, processors are assumed to correspond to separate distributed collections of system components. In other words, given the final system realization and any one of the processor structures present in the hierarchy, one should be able to uniquely partition all

system components into equivalence classes and to establish a meaningful one-to-one correspondence between these equivalence classes and the entities (processors and interconnections) of the chosen processor structure.

The PROCESS/PROCESSOR ALLOCATION captures the distribution of the processes among the available processors. In its simplest form, the allocation may be static, i.e., does not change during the execution of the system. In such cases, all processes are partitioned among the available processors with the links being partitioned accordingly between processors and their interconnections. Reliability, workload balancing, and other design considerations, however, often require dynamic changes in the allocation of processes and links among the available processors and interconnections. (Note: An additional degree of complexity may be noticed in systems which permit a process to be mapped simultaneously on several processors. This occurs, for instance, when the code associated with a particular realization of some process and the execution of the corresponding instructions are the responsibilities of two separate processors. The definitions from Appendix E do not rule out such cases.) The separation of the allocation/reallocation issue from the functional details of the process structure has the potential to significantly reduce the complexity of analyzing both the individual subsystems and their relationships.

The PERFORMANCE SPECIFICATIONS deal with the performance attributes of the system and with the models used to relate the performance attributes to the selected system architecture and to each other. A performance attribute may be associated with either the process or the processor structure and represents either a performance requirement originating with some performance constraint or a performance characteristic that has been established to be true, i.e., it was validated. Performance requirements (i.e., constraints) are assumed to propagate top-down from the process structure to the processor structure, and from one subsystem to the next. The performance characteristics, however, propagate bottom-up; only when the exact characteristics of the processor structure are known one may deduce with certainty the characteristics of the process structure. Moreover, an acceptable design demands that all performance characteristics imply the satisfiability of the corresponding performance requirements. In this context, performance models assume a dual role. First, they assist one in determining the performance requirements of the processor structure from those of the process structure. Second, they propagate the performance characteristics of the processor over the process structure.

The BINDING OPTIONS represent a non-empty (possibly infinite) set of system realizations that are still feasible at a given point in the design process. This set is very large at the start of the system architecture design phase and, through successive design refinements, is systematically reduced to a manageable size upon entering the binding phase. Because at no point in time it is possible to enumerate the members of this set, the designer specifies it indirectly via a distinguished category of constraints called binding constraints. They are formulated during the design process as a result of explicit design choices (which rule other out alternatives) and due to conclusions drawn from various design studies

and analysis of the stated system requirements, available technology, anticipated operating environment, etc.

Finally, the CONSTRAINTS that appear in each design specification are inherited from the original system requirements and carried along throughout the entire design. Different constraints, however, affect the design at different points in time. Some represent the origin of the performance requirements while others may affect certain aspects of binding. It is the designer who brings into consideration the appropriate constraints at the right place in the design.

3.4.4 INFORMAL DEFINITION OF HARDWARE/SOFTWARE REQUIREMENTS

The hierarchy of design specifications present in the processing model is mapped during the binding phase into off-the-shelf, customized, and custom-made software and hardware. Separate software requirements specifications are generated for each subsystem. In addition, hardware requirements specifications are produced for the lowest level subsystem in the processing model hierarchy. While there is great variability in the way in which both software and hardware requirements need to be specified, they generally include the following:

- a specification of the functional and performance requirements of the hardware or the software (present, for the most part, in the respective design specification);
- a specification of all relevant interfaces (between subsystems, between components residing on different machines, between components developed separately, etc.);
- a mapping from parts of the proposed design onto existing hardware or software;
- a list of existing hardware or software to be used.

A simple inventory system may be used to illustrate the nature of the hardware/software requirements:

SOFTWARE REQUIREMENTS.

LEVEL 1 (Application Program).

- functionality given by an inventory control language whose syntax and semantics have been fully specified; no performance constraints;
- user interface via the inventory command language; access to the database defined by the INGRES user manual; the implementation language C;
- all database manipulations are relegated to INGRES;
- off-the-shelf software to be used: INGRES -- a relational database package.

LEVEL 2 (Operating System).

- functionality given by the UNIX user manual;
- user interface via UNIX standard commands; UNIX version supported by the PDP 11/40 machine and compatible with INGRES;
- no changes or enhancements to the UNIX operating system permitted;
- off-the-shelf software to be used: UNIX operating system.

HARDWARE REQUIREMENTS.

LEVEL 2 (Hardware Configuration).

- the hardware configuration consists of a PDP 11/40 with 64k bytes of main memory, a VT52 compatible CRT terminal, a 1200 baud printer, and two disk drives for 2.5 megabytes disk cartridges;
- one serial port for interfacing the crt and the processor; one parallel port for the printer; two direct memory access ports for the disk drives;
- the mapping of functions to components is trivial in this case;
- specific printer, crt, and disk drives could be listed here.

The relation between the processing model and the hardware/software requirements is further analyzed in the next section.

3.4.5 METHODOLOGY OUTLINE

3.4.5.1 GENERAL REMARKS

This section discusses a proposal for a class of TSD Methodologies focused on the design activities leading to the identification of the hardware and software components in distributed systems. The presentation starts with a statement of objectives. It is followed by the design strategy to carry out the system architecture design phase. The strategy covers both the design of the individual subsystems and the sequencing of design activities between subsystems. Finally, the discussion turns to a systematic way of accomplishing the task associated with the binding phase.

The principal goal of the proposed methodology is to increase the quality and productivity of the design of large distributed systems. Reaching this goal, however, places the following demands on the nature of the TSD Methodologies:

- an ability to explore in a systematic manner a large design space by separating system level issues from those involved in the design of hardware and software and by placing the selection of hardware and software (i.e., hardware/software trade-offs) on a more rational base than it has been done in the past;
- a structuring of the design process in a way which assures a great degree of control over design complexity and promotes

incremental verification of both the functional and performance aspects of successive system design refinements;

- a strategy which is based on general design principles rather than the peculiarities of a specific class of applications but which, at the same time, is adaptable, i.e., may be tuned to a given application.

It is our contention that the design strategy we have selected indeed has all these attributes. The remainder of this section describes the proposed strategy while Sections 3.5 and 3.6 illustrate the TSD Methodologies' ability to adapt to the specifics of several very diverse applications. The ultimate validation, however, has to come from empirical studies in which the methodologies are applied to specific problems. Furthermore, specification languages and an appropriate assortment of techniques need to be developed in order to provide the designer with a computer aided environment that would assure the high productivity to which the TSD Methodologies aspire.

Before presenting the methodology it is necessary to point out that, for the sake of clarity, certain simplifying assumptions are being made throughout Section 3.4.

- design backtracking due to errors receives limited coverage;
- parallel development of portions of the design by different teams on the project is ignored despite the great opportunities for concurrency within a project;
- most project management activities are omitted;
- system integration is not discussed.

While they do not alter the overall flavor of the strategy, they may make the methodology appear somewhat inflexible. We hope that by pointing them out early in the presentation, the reader will have no trouble in discerning the difference between the overall design strategy and the artifacts of the simplifying assumptions.

3.4.5.2 SINGLE SUBSYSTEM DESIGN STRATEGY

As indicated earlier, systems are described in terms of a hierarchy of design specifications. They force a structuring of the system in terms of a number of subsystems, each supporting the subsystem above. The methodology requires the design of individual subsystems to proceed top-down. Within the general context of top-down design, however, several related activities are interleaved (in the manner specified in Section 3.4.6). These design activities are outlined below.

Successive and concurrent refinement of both the process and the processor structures. The fact that a given system function may be decomposed in more than one way is well-known. This design freedom is not a menace, as seen by some (e.g., [BERG81]), but rather a degree of flexibility essential to good design. The selection

between alternate decompositions is not intrinsic to the decomposition itself, but depends upon of the designer's objectives (maintainability, clean abstraction, simple interfaces, reliability, etc.). Among them, the availability of certain (existing or postulated) means of support may also affect the functional decomposition. The case when the process structure is affected by earlier choices of the processor structure is illustrated by the way in which the solution for a certain computational problem may take different forms if one assumes the use of a high speed minicomputer with or without an attached array processor. The converse situation (which occurs frequently in the data processing field) is where the needed hardware is selected based on the result of a functional decomposition of the application problem, where the decomposition is guided by some modularization principle.

Because of this interdependence between the selection of the process structure and of the processor structure, TSD Methodologies emphasize the concurrent refinement of both structures. While accommodating the special cases where the peculiarities of the application force one structure or the other to be dominant, this approach offers the system designer the added flexibility required by an unbiased treatment of the hardware/software trade-offs problem. Furthermore, the balance is allowed to shift in one direction or another, not due to personal prejudices, but due to constraints that affect the range of acceptable system realizations.

Top-down propagation of performance requirements. Fundamental to the conception of the TSD Methodologies is the assumption that performance constraints direct to a large extent the designer's activities. Performance requirements recognized at the top level of a design specification propagate from one level to the next through the assumptions the designer makes at one level about the characteristics of the next. The assumptions later become requirements and the cycle continues. In order for the designer to make reasonable assumptions, two things are needed: past experience and adequate performance models that relate the presumed performance characteristics of entities of some level and the performance requirements placed over the particular level of the design specification. The nature of the performance models has to change according to the level of functional detail. When the level of abstraction is high, the models are less detailed, less accurate, and also less costly than when lower levels of the specification are reached. The scheme has two advantages. On one hand, it allows performance considerations to influence design decisions early on. On the other hand, it holds the promise that this may be achieved in a cost effective manner. (This idea has received some endorsement in recent publications [KUMA80, SANG79].)

Bottom-up propagation of performance characteristics. While the performance requirements flow top-down, the validated performance characteristics (once available) propagate in the opposite direction [BOOT80]. The use of the performance data is important in making immediate readjustments of the subsystem design and establishes the accuracy of the assumptions that were made and the feasibility of the

proposed design. (The way in which the performance characteristics become available is discussed later.)

Binding constraints accumulation. The hardware/software trade-offs dynamics manifest themselves during the system design stage as a gradual narrowing down of the range of feasible realizations, i.e., binding options. This takes effect through a growth in the set of recognized binding constraints. The set, originally inherited from the subsystem above, is augmented from several sources. First, each design decision taken (e.g., successive refinements, allocation, etc.) rules out all realizations which have adopted different approaches. Second, design studies that look ahead to low level but potentially difficult components of the system also affect the directions the designer is willing to consider. If, for instance, there were no totally distributed concurrency coordination algorithms, a database design based on their potential availability would have to be discarded. Third, inference studies may suggest that the use of some technological alternatives may be unfeasible (due to their impact on other aspects of the system or on its operation environment, etc.) or, although feasible, not recommended (due to anticipated technological trends, for instance). Finally, the availability of certain software or hardware may dictate a design solution which takes advantage of such off-the-shelf components in order to reduce development costs.

Systematic error detection. Error detection is supported via a number of checks placed at various critical points in the sequence of design activities within the tasks/subtasks and in the tasks/subtasks review sections of the methodology specification. They involve consistency checks between adjacent levels of a design specification and between related components of the specification (e.g., process/processor allocation versus the process and the processor structures). The checks also include logical verification between the design specification of one subsystem and its requirements which, in general, are established by the specification of the subsystem above, if any. For the top subsystem in the hierarchy, however, the subsystem requirements are the same as the system requirements. This issue is considered again in the discussion of the subsystems' design dependencies which follows.

REFERENCES

- [BERG81] Bergland, G. D., "A Guided Tour of Program Design Methodologies," Computer 14, No. 10, pp. 13-37, October 1981.
- [BOOT80] Booth, T. L. and Wiecek, C. A., "Performance Abstract Data Types as a Tool in Software Performance Analysis and Design," IEEE Trans. on Soft. Eng. SE-6, No. 2, pp. 138-151, March 1980.
- [KUMA80] Kumar, B. and Davidson, E. S., "Computer System Design Using a Hierarchical Approach to Performance Evaluation," CACM 23, No. 9, pp. 511-521, September 1980.

[SANG79] Sanguinetti, J., "A Technique for Integrating Simulation and System Design," Proc. Conf. on Simulation, Measurement and Modeling of Computer Systems, pp. 163-172, August 1979.

3.4.5.3 OVERALL SYSTEM DESIGN STRATEGY

The structuring of the system design in terms of the proposed hierarchy of design specifications, is motivated by the desire to control complexity through a systematic and strict separation of concerns. The idea originates in part with the already common concepts of virtual machine and stratified design (layers of virtual machines [ROBI77]). When using a programming language, the designer is not in the least concerned with the implementation details of the language (if the language is properly designed). Similarly, when working at one level of a stratified design the designer deals only with the semantics of the operations available at that point and not with their possible realizations. The TSD Methodologies attempt to exploit this approach in the context of distributed systems by adapting it accordingly.

The designer starts from the system requirements and, through successive refinements of the process and processor structures, defines both the way in which the functionality specified in the conceptual model is implemented and the support needs for such an implementation (e.g., message exchange capability, process reallocation due to failures, storage management, etc.). The top design specification is said to describe the application subsystem, due to the nature of its functionality which is directly relevant to the application at hand. All subsequent specifications are said to describe support subsystems.

As already stated, the construction of each design specification takes place in a top-down manner. However, it is often the case that, prior to completing the specification, the support needs required by the process structure may become clear. In such cases, the generation of the current design specification may be temporarily suspended and the design of the supporting subsystem may proceed. Despite the fact that the strict top-down design strategy could be followed, the designer may choose to move to the next subsystem in the hierarchy in order to minimize the risk that some of the assumptions made about the support subsystem may prove to be wrong. However, once the designer decides to move to the subsystem below, the design discipline prescribed by the TSD Methodologies requires one to complete the design of the support subsystem prior to resuming the design of the subsystem above. This reduces thrashing between subsystems and enables the designer to make use of the performance characteristics of the support subsystem in the adjustment and completion of the specification for the subsystem being supported.

The diagram that follows depicts the result of applying this strategy to the design of a computer graphics system. Design activities are represented by groups of slashes. The left most column of slashes corresponds to the design of the top subsystem, i.e., the application subsystem.

```
/ design of
/ graphics
/ language
/
// design of
// graphics
// language
// interpreter
//
/// design of
/// graphics
/// and
/// communication
/// primitives
///
//// design of
//// graphics
//// hardware
////
///
//
```

Aside from the global sequencing of design activities, an understanding of the role that the processing model plays in the system design stage requires the definition of three important concepts. The first one is the notion that the top design specification (the application subsystem) implements the system requirements. The second is the support relation between the design specifications within the processing model hierarchy. Finally, the concept of superficial binding makes the transition to the binding phase.

A design specification is said to implement the system requirements when its process structure is logically equivalent to the functionality captured in the conceptual model. The definition may be actually extended to the individual levels developed during the top-down design of the specification. A given level in the specification implements the system requirements if its process structure is logically equivalent to an abstraction of the conceptual model. These definitions establish the correctness criteria to be employed during the design of the application subsystem and form the foundation for future automated checking of the top design specification.

In the most basic terms, "subsystem B supports subsystem A" implies two things about B: (1) it contains the design of functions (unrelated to the application area) which were assumed to be available during the design of subsystem A and (2) it may represent a further refinement of the degree of distribution within the system. With regard to the first role of a support subsystem, it must be pointed out that the ultimate realization of the support relationship may assume a great variety of forms. Consider, for instance, the special case when both subsystems are eventually implemented in software:

- the programs of A may actually invoke the programs of B either as procedure calls or as macros -- such is the case when B realizes the communication protocol assumed by the message sending and receiving commands used by A;
- the programs of A may be interpreted by programs in B -- the availability of a LISP interpreter may be one of the support functions assumed by A;
- the programs of A may be objects (i.e., data) manipulated by programs in B -- programs in B may have the responsibility to monitor and relocate the programs of A in case of equipment failure or for load balancing purposes.

About the potential increase in the degree of distribution from one subsystem to the next, the idea may be rendered easily through the use of the earlier graphics system example.

DESIGN ACTIVITY

PROCESSOR STRUCTURE TOPOLOGY

/ design of
/ graphics
/ language
/
/

(user) U <--> X --> I (image)

```
// design of
// graphics
// language
// interpreter
//
//
```

U \Leftrightarrow Y1 \Rightarrow Y2 \Rightarrow I

```
/// design of
/// graphics
/// and
/// communication
/// primitives
///
///
```

U \longleftrightarrow Z1 \rightarrow Z2 \rightarrow I

```
//// design of
//// graphics
//// hardware
////
```

-----> W22 --> I

U <--> W1 --> W21
 where
 W1 = minicomputer
 W21 = image buffer
 W22 = display unit

///
///
///
//
//
//
/
/
/
/

The third important concept, superficial binding, must be considered in conjunction with the binding strategy.

REFERENCES

[ROBI77] Robinson, L. and Levitt, K. N., "Proof Techniques for Hierarchically Structured Programs," CACM 20, No. 4, pp. 271-404, April 1977.

3.4.5.4 BINDING STRATEGY

A processing model is considered bound when all its entities are mapped into software and hardware to be obtained (some by purchasing off-the-shelf components, others by customizing available components, and yet others by custom building them). The binding process (also called hardware/software trade-offs) starts in the system architecture design phase and reaches its conclusion in the binding phase. In the first of these two phases the growth in the set of binding constraints (explained earlier in this section) reduces the set of feasible alternatives, thus biasing the design toward certain technological alternatives the designer considers to be most promising. This biasing becomes very strong when the designer chooses to structure the system around the potential use of available components; the system entities tentatively associated with such components are said to be superficially bound to them. Note that one entity may be superficially bound to one or more alternatives.

At the point when the binding phase is entered, large parts of the processing model may be superficially bound. The strategy used to accomplish the binding could be called a "most constrained first" approach. The designer starts by identifying binding alternatives for the most constrained areas of the specification. This results in the immediate generation of new binding constraints over the remaining parts of the design which, in turn, eliminates from consideration many fruitless alternatives. Even if one is careful to always limit the investigation to a tractable number of alternatives, the total number of system configurations being evaluated at one time could grow rapidly. If, for instance, one needs to merely select three machines and there are four alternate candidates for each, the total number of system configurations reaches sixty-four. While some configurations may be ruled out by incompatibilities between some candidates associated with areas of the design which are interfaced to each other, the designer needs to weed out many more by employing guidelines such as cost minimization, maintainability, uniformity, etc. Once the entire specification is superficially bound to several alternate configurations, their number needs to be reduced to one by evaluating the weak and the strong points of each of them. Now the system specification is bound.

A last task still to be carried out is the generation of the software and the hardware requirements. They have to include such things as the functionality of various components, performance and other constraints, interface definitions, etc. The exact contents and form of these requirements is hard to formalize due to significant variability between systems. This concludes the informal presentation of the design strategy proposed for the system design stage.

3.4.6 FORMALIZATION OF THE DESIGN STRATEGY

NOTE: The flow of control constructs employed in this section are explained in Appendix D. Tasks, subtasks and procedures should be treated like recursive procedures present in common programming languages such as PL/1 or Pascal with the special provision that their definition appears at the place of their first invocation. Consequently, at the place of definition and first invocation, parameters are defined and initialized at the same time.

Furthermore, all simplifying assumptions explained earlier are reflected in the way the strategy is formalized here.

TASK System-Architecture-Design.

SUBTASK Subsystem-Design(*i*=1).

Review subsystem requirements (for *i*=1 the subsystem requirements correspond to the system requirements and the subsystem is called the application subsystem; otherwise, the requirements are given by the processor structure definition and process/processor allocation defined by the subsystem (*i*-1)).

Set the set of binding constraints to be the same as the binding constraints of subsystem (*i*-1), unless *i*=1, in which case the set of binding constraints starts by being empty.

Identify those technological alternatives that may be ruled out as unacceptable and/or limit the set of technological alternatives only to those that appear to be appropriate; formulate constraints which would reflect these considerations; add these constraints to the binding constraints.

IF the subsystem *i* is already available THEN DONE.

Develop top-level (i.e., level 1) for the design specification of the subsystem *i* based on some abstraction of the requirements definition; the process structure includes the modelling of the subsystem's environment; the processor structure topology is inherited from the subsystem (*i*-1), if it exists.

PROCEDURE Subsystem-Refinement(*j*=2).

```
{ ==> { Generate the process structure for level j by decomposing or
      by copying the process structure of level (j-1).
      Generate the processor structure for level j by decomposing
      the processor structure of level (j-1) w.r.t. the needs of
      the process structure on level j. } ;
==> { Generate the processor structure for level j by decomposing or
      by copying the processor structure of level (j-1).
      Generate the process structure for level j by decomposing the
      process structure of level (j-1) w.r.t. the capabilities of
```

the processor structure on level j.}}

Define process/processor allocation on level j; the allocation may be static or dynamic and must be consistent with the allocation rule used by the subsystem (i-1), if it exists.

iteration:

```
LOOP{ ==> Adjust the specification of the level j. |
      ==> Propagate both process and processor performance
          requirements of level (j-1) over the level j and refine the
          performance models used at level(j-1); analytical,
          simulation and empirical techniques may be required to
          support the requirements propagation activity. !
      ==> Investigate inference issues related to decisions taken at
          this level and eliminate binding options that are shown to
          be inappropriate. !
      ==> Superficially bind aspects of the subsystem to already
          available software/hardware, if such decisions are strongly
          motivated by constraints or design principles. !
      ==> Carry out logical and consistency checks for level j. !
      ==> Carry out design studies for this or subsequent
          subsystems. !
      ==> BREAK. }
```

IF level j does not refine correctly level (j-1) THEN BACK.

IF level j is not an implementation of some abstraction of the subsystem requirements THEN BACK.

```
{ Process and processor structures are not completely refined
  ==> INVOKE Subsystem-Refinement(j+1). !
```

Processor structure is completely refined
==> { INVOKE Subsystem-Design(i+1).

```
LOOP{ ==> Propagate the performance characteristics of the
      subsystem (i+1) to the processor structure of the
      subsystem i and, subsequently, to the process
      structure of subsystem i. !
      ==> Adjust the specification of subsystem i. !
      ==> BREAK. }
```

IF process structure is not completely refined THEN
{ PROCEDURE Finish-Refinement(jf=j+1).

Generate the process structure for level jf by
decomposing the process structure of level (jf-1) w.r.t.
the capabilities of the processor structure on level jf
(same as on level (jf-1)).

iteration:

```
LOOP{ ==> Adjust the specification of the level jf. !
      ==> Propagate process structure performance
          requirements of level (jf-1) over the level jf
```

through the use of appropriate performance models. ;
==> Investigate inference issues related to decisions taken at this level and eliminate binding options that are shown to be inappropriate. ;
==> Superficially bind aspects of the subsystem if such decisions are strongly motivated by constraints or design principles. ;
==> Carry out logical and consistency checks for level jf. ;
==> BREAK.}

IF level jf does not refine correctly level (jf-1) THEN BACK.

IF level jf is not an implementation of some abstraction of the conceptual model THEN BACK.

IF process structure is not completely refined THEN INVOKE Finish-Refinement(jf+1).

PEND.})}

PEND.

STREVIEW.

F(Check the self-consistency of the design specification for the subsystem i.) ==> BACK.

F(Perform logical verification of the design specification with respect to its functional requirements.) ==> BACK.

F(Check that all performance constraints placed on subsystem i are met, given the characteristics of subsystem (i+1).) ==> BACK.

F(Determine that all consequences of the proposed design are acceptable.) ==> BACK.

STEND.

Develop system testing plan.

TREVIEW.

F(Evaluate the system testing plan.) ==> BACK.

TEND.

* * *

TASK Binding.

LOOP{ IF the system is superficially bound THEN BREAK.

Identify those design entities and groups of design entities which are not superficially bound and have fewest degrees of freedom with respect to binding.

FOR all such entities and groups DO
{ // { Identify binding candidate selection rules.

Select tractable set of candidates.

Establish the mapping between the candidates and the related design entities.}}

Define the compatibility relation between the candidates associated with various parts of the design.

IF Compatibility problems are found THEN BACK.

Keep a reduced list of compatible alternatives based on various guidelines such as cost minimization, uniformity, flexibility, interface complexity, etc. }

Evaluate the possible system configurations and reduce their number to one.

{ Generate software requirements including: functionality; explicit statements with regard to both constraints and degrees of freedom; the specifications of the interfaces between the components of each subsystem, between subsystems, and with the hardware; and the off-the-shelf and customized software to which some of the components are bound. //

Generate hardware requirements including: functionality; explicit statements with regard to both constraints and degrees of freedom; the specifications of the interfaces between the hardware components and with some of the software; and the off-the-shelf and customized hardware to which some of the components are bound. }

Develop integration plan.

TREVIEW.

F(Check the self-consistency of the software requirements.) ==> BACK.

F(Check the self-consistency of the hardware requirements.) ==> BACK.

F(Check consistency between hardware and software requirements.) ==> BACK.

F(Verify the functional aspects of the hardware/software requirements against the processing model.) ==> BACK.

F(Check that all performance constraints placed on the system are met, given the characteristics of the hardware and of the software.) => BACK.

F(Determine that all consequences of the proposed hardware/software selection are acceptable.) => BACK.

F(Evaluate the integration plan.) => BACK.

TEND.

* * *

3.5 DISTRIBUTED REAL-TIME SYSTEMS ILLUSTRATION

3.5.1 INTRODUCTION

In the most general sense of the term, a "real-time" application is one which places stringent demands on computer system response time. More often, however, the term refers to an application in which the computer system is part of the control loop for some other system. In this control capacity, the loop delay attributable to the computer system must be very small compared to the rate at which the controlled system can change state. This is the sense in which we use the term here.

The need to meet stringent response time requirements makes the task of system design more difficult. The degree of difficulty depends on the complexity of the processing and on the time allowed to do the processing. For some application areas, the response-time constraint can be met by an unsophisticated program running on a typical off-the-shelf microprocessor. For other areas, the demands are so severe that they require the devising of novel algorithms that distribute the processing over collections of processors working in parallel.

When the control system is located remotely from the system being controlled, communication delays contribute to the loop delay. This reduces the amount of time available for the system to compute its response and, as a consequence, makes the design task more difficult. When the control system is embedded within the system being controlled, the communication delays are small and the time available for system response is maximized. However, embedded systems must live within the physical environment of the controlled system, and this operating environment can impose constraints that greatly increase design complexity.

Consider, for example, the control of a guided missile. If the control system is embedded in the missile, the design must meet severe volume, weight, and power constraints, and must be able to withstand the g-loading, vibration, and other stresses peculiar to that operating environment. These factors disappear if the missile is remotely controlled, but at the cost of having to deal with communication delays and the risk of communication disruption.

The trade-offs between remote and local control are important design issues whose relative merits are weighed during the problem identification phase of the system design process. The trade-off consideration is not a matter of choosing one over the other, but rather, deciding which aspects of system control should be handled remotely and which should be embedded.

A good example of this is provided by the unmanned space probe that recently flew by Jupiter and Saturn and is now on the way to Uranus. Embedded control takes care of the minute by minute operation of the space craft, while earth-based stations interact with the craft for purposes of defining future activities. This arrangement is mandated by the fact that communication delays between Earth and craft get larger as the craft moves farther away. Since communication delays on the order of minutes occur

early in the mission, total control of the craft from Earth is not just difficult, it is impossible.

In summary, then, a characteristic common to all real-time applications is the need to meet stringent response time constraints. An important subclass, the embedded systems, must also meet constraints imposed by their operating environment. In application areas such as weapons systems and medical prostheses, the design of these systems will often push the state of the art. Solutions can require the design of custom hardware, the development of novel processing architectures, or the devising of new computational algorithms. The most demanding cases will require all three.

The impact of these considerations on design methodologies is clearly pronounced. To be effective, a methodology must facilitate the evaluation of correctness from both the functional and performance standpoints. These evaluations must be made during the problem definition stage in order to assure that the design specifications are indeed adequate for the intended control application, and they must be made during the design process in order to assure correctness of the design. For those applications that require the design of custom hardware, these evaluations must also be carried out at the actual hardware/software level in order to assure correctness prior to the costly process of hardware design and manufacture. The sophistication of the evaluation tools will depend on the class of system being designed. Relatively simple tools will suffice for some application areas while other areas will require emulation facilities in order to perform the evaluations within a realistic time period.

Because real-time systems are strongly constrained, a methodology must provide formalisms for expressing function/constraint relationships in a precise, unambiguous manner. This applies to both the specification of system requirements and the description of system design. Additional expressive capabilities must be provided for specific application areas. Among the more common auxiliary needs are abilities to express the following: asynchronous interactions at the system interface; processing structures comprised of concurrent, communicating processes; the dynamic allocation of processes to processors.

Finally, design strategies for real-time systems are driven by the need to meet constraints. The implications of this depend on the class of system being designed. For some application areas, it means making a few performance-related adjustments to a processing model developed through straightforward functional decomposition. However, for areas with constraints such as volume/weight limitations, a need for high throughput and fault tolerance, etc., there may be no way to adjust a straightforward processing model to meet these constraints. In these cases, an appropriate processing model must be developed around general techniques for meeting the various constraints and around the capabilities of current hardware technology.

The diversity of methodological needs makes it difficult to devise a methodology that is well-suited to all real-time systems. The more realistic approach is to specialize methodologies to particular

application areas, optimizing them to the particular needs of the area and to the backgrounds of design personnel working in that area. The strong impact of application area on design methodology is illustrated in the next section. It gives an overview of a particular application area, summarizes the impact of this area on methodology needs, and reviews the design of a particular system within that area.

3.5.2 BALLISTIC MISSILE DEFENSE SYSTEMS

This section begins by giving an overview of a broad application area, Ballistic Missile Defense Systems, and discussing the impact of area needs on system requirements. The implications of these requirements on design methodology is then considered, with attention being narrowed to a subarea, homing interceptors, in order to make the discussion more concrete. A processing model typical of systems in this subarea is presented and is illustrated for an actual system, the Modular Missile Borne Computer.

3.5.2.1 INTRODUCTION

Ballistic Missile Defense (BMD) systems are military systems used to detect and to defend against missile-based attacks. Detection is based on active (radar) and passive (optical) sensor systems, while defense is based on the use of interceptor missiles. There are two distinct operational phases for these systems. The precommit phase deals with the gathering and analyzing of sensor data and the maintaining of battle ready status. The postcommit phase deals with the launch and targeting of interceptors and with all other aspects of battle management.

Computer systems are used extensively in BMD systems. Computer systems embedded in the sensor systems direct the sensors, preprocess the sensor data, and effect communication with battle management systems. Computer systems embedded in the interceptor missiles perform target tracking (via onboard sensors), navigation, guidance, and communication with battle management systems. Computer systems in the battle management systems process the data received from the sensor systems, schedule the targeting and launch of interceptors, monitor the effectiveness of each defensive action, and determine successive defensive actions for as long as the battle lasts.

The nature of the BMD mission imposes tough requirements on most of its computer systems. First, because of the limited time in which to detect a threat and intercept it, response time requirements are severe. Second, because the processing needed for various discrimination and tracking functions requires complex mathematical computations on large volumes of data, throughput requirements are severe. Third, the critical nature of the BMD mission demands high availability. Fourth, the changeable nature of defense requirements requires that systems be easy to upgrade. Fifth, airborne and spaceborne systems must meet severe volume/weight constraints. Sixth, systems must withstand radiation, shock, and electromagnetic stress associated with an attack. In addition, missile-borne and satellite-borne systems must withstand the stresses of

launch and the stresses of harsh operating environments.

These requirements have a significant impact on the design of BMD computer systems. For most systems, there is no way to meet either the response time requirements or the throughput requirements with a uni-processor architecture. Multi-computer architectures are generally required and, for systems with volume/weight limits and harsh operating environments, this usually means custom, state-of-the-art hardware. Fortunately, the mathematical nature of most of the processing is well understood, including the opportunities for concurrency, and this facilitates the creating of processing models suited to such architectures.

Most BMD systems handle a variety of tasks, each with specific operating requirements. Some tasks are driven by the arrival of data which may occur at a uniform rate or randomly. Some are driven by the passage of time, such as issuing reports at specific times and transmitting data at specific rates. Others are driven by exceptions such as the detection of an abnormal system condition. The management of these tasks usually requires the services of a real-time system executive. This executive provides intertask communication and coordination, and schedules tasks and allocates resources on the basis of input events, exceptions, current time, task priority, task temporal requirements, and precommit/postcommit status.

The extreme importance of system availability means that most BMD systems are fault tolerant to some degree. Since methods for achieving fault tolerance depend on hardware and software structure and on the type of faults to be dealt with, the physical and logical structures of these systems are strongly influenced by the hazards of their specific application.

3.5.2.2 METHODOLOGICAL IMPLICATIONS

The BMD application area has a variety of sub-areas that warrant separate treatment. These include the control of active sensors, the control of passive sensors, the control of homing interceptors, and battle management. Each of these application areas has distinctive processing requirements which require different process structures. They also differ in operating environment and possibilities for repair. For example, battle management systems are typically earth based, are not subject to volume/weight limitations, and operate in non-hostile environments. They can thus utilize large mainframe architectures and commercially available hardware and software, and they can be repaired through manual intervention. In contrast, embedded systems that control homing interceptors are subject to volume/weight limitations and operate in a hostile environment. They require special hardware and software, and special techniques for dealing with faults. In addition, the nature of their processing tends to be more rigidly defined and less subject to change. These differences have a considerable impact on system design and warrant different forms of design methodology.

Listed below are the methodological implications of applications dealing with the control of homing interceptors. The characterization is at a relatively high level, in terms of the gross structure of the processing model and the general characteristics of the methodology components.

PROCESSING MODEL

VERTICAL STRUCTURE. The processing model has a two tier vertical structure consisting of an application oriented tier supported by an executive tier.

PROCESS STRUCTURE. The process structure for each tier consists of a collection of concurrent communicating processes, many of which are event driven. Because of the need to meet stringent temporal performance requirements, the processes of both tiers are tightly coded, primarily in terms of hardware primitives.

PROCESSOR STRUCTURE. The processor structure that supports the system is a locally distributed multi-computer network. This structure develops in an incremental manner, the gross structure being defined by the needs of the application tier, and the fine structure being defined by the needs of the executive tier.

METHODOLOGY CHARACTERISTICS

SPECIFICATION FORMALISMS. Specification formalisms must be able to express the concurrent, multi-task, event-driven nature of both tiers.

TOOLS. Sophisticated evaluation tools are needed for verifying functional and performance correctness. Simulation at the object code level is needed to prove the suitability of the hardware/software mix.

DESIGN STRATEGY. There is an intrinsic need for fault tolerance. Because of the dependence of this property upon the logical and physical structure of the system, the design strategy is oriented around techniques for achieving this property.

Although a working methodology with these characteristics does not currently exist, there have been substantial research efforts on various components. For example, TRW, a company that is a major BMD contractor, has developed a requirements statement language called RSL [BELL76] for specifying the functional, temporal, and analytic requirements of real-time tasks, and has developed a computer-aided system called SREM [ALFO77] for developing, maintaining, and analyzing RSL system specifications. TRW is also developing a computer-aided system called FAST [McCL75] for the simulation and analysis of computer systems. As a part of the FAST program, a high order computer description language

called SMITE [SMIT77] has been developed for programming diagnostic emulations, and a SMITE compiler has been developed for the Nanodata QM-1, a horizontally microprogrammable machine suited to emulation applications.

The next section discusses the design of a specific interceptor control system. The purpose is to illustrate the suitability of the above processing model and to show how strongly the design strategy is driven by availability considerations.

3.5.2.3 EXAMPLE: THE MODULAR MISSILE BORNE COMPUTER (MMBC)

MISSION

An intelligent interceptor missile is one that provides its own target tracking and guidance through the use of onboard sensors and an onboard computer system. The Modular Missile Borne Computer (MMBC) is a computer system designed specifically for this application. A good description of the design objectives and the details of the MMBC system are given in [RAMS79, APPL79, KINN79, ARN079] and our presentation here is based on that material.

The mission responsibilities of the system are as follows. At launch time the system is given information about particular incoming threat objects and given a battle management strategy. From that point on the system functions autonomously. It acquires and maintains track on the assigned threat objects, recognizes any new targets deployed by the threat objects, performs detailed discrimination on any undiscriminated objects and begins tracking identified re-entry vehicles, issues guidance, navigation, and control commands necessary to accomplish intercept, and finally initiates homing and fuzing.

Information needed to control the missile during the mission is obtained by processing image data from onboard optical sensor arrays. Because of the speeds at which the missile and the targets move, image data is acquired at a high rate in order to remain abreast of the situation and make the necessary course corrections. This data rate ranges from 10^{**6} to 10^{**9} words per second, the actual rate depending on the particular sensor configuration being used. These high data rates, coupled with the amount of processing required per input word, require the MMBC to have a multi-computer structure.

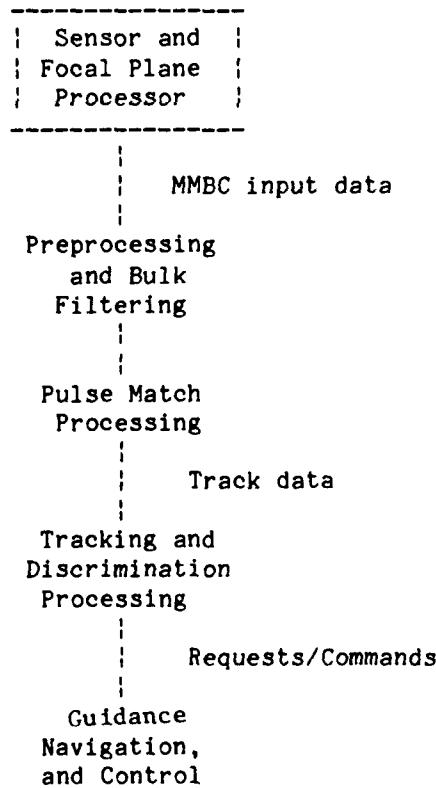
Because of the volume, weight, and power limitations associated with being embedded in a missile, the MMBC computing components are microcomputers, and, because of mission stresses such as g-loading, vibration, and temperature extremes; radiation from terrestrial, solar, and celestial sources; and shock and radiation from nuclear detonations; the computing components are assembled from custom hardware. Careful design, manufacture, and selection reduce the risk of hardware failure, but there are limits as to what can be achieved. As a result, the MMBC is also designed for fault tolerance.

VERTICAL STRUCTURE

The MMBC has a two-tier vertical structure. The top tier contains the application software while the second tier contains the system executive. This two tier structure is not just a conceptual model, it is implemented in the system in a very strict manner. There are several reasons for this, but the primary reason is that the MMBC is to be used in a variety of applications, each needing different configurations of hardware and different application processes. The executive provides a virtual machine that allows the application code to be configuration independent.

PROCESS STRUCTURE

A high-level view of the application tier process structure is shown below. The Sensor and Focal Plane Processor acquire image data and prepare it for transmission to the MMBC. The MMBC converts this data into track data and uses that to generate the commands needed for Guidance, Navigation, and Control. Each of the first three processes are implemented by a collection of concurrent tasks executing concurrently on separate processors. These tasks are organized in a manner that promotes throughput and fault tolerance -- for details see discussion of static masking under heading "Design Strategy".

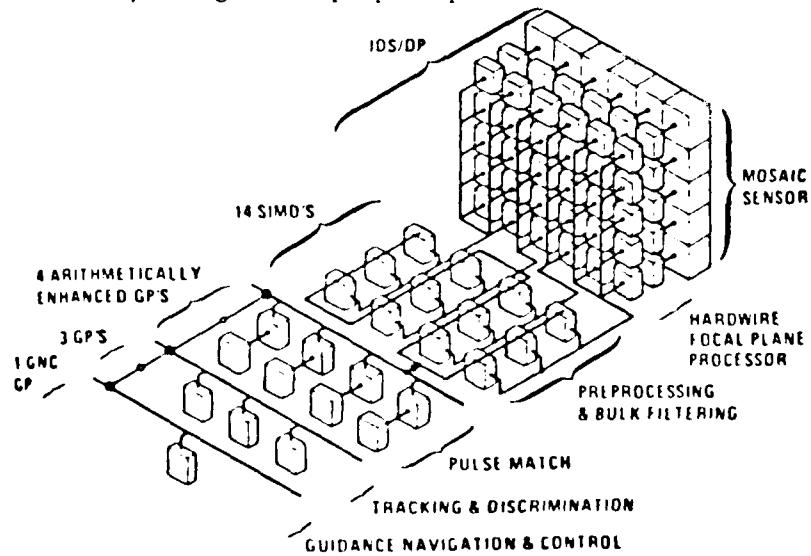


The application processes execute concurrently and are coded in an instruction set consisting of hardware primitives (PDP11-like) and instructions that trap to the executive. Communication between processes is message based, with the destination process being specified by logical name. The task of getting the message from the sending process to the receiving process is taken care of by the executive and is transparent to the application software.

The executive tier process structure consists of a local executive for each processor and a link interconnecting all local executives. No system-wide scheduling or resource management is attempted. Each local executive consists of a set of concurrent processes for performing scheduling and dispatching, interrupt and trap handling, memory management, communications management, run-time management, miscellaneous system services, instrumentation and performance monitoring. Commonly used functions such as global bus allocation, message assembly, linked list manipulation, context saving, and procedure entry/exit are implemented in hardware while less commonly used functions are implemented in software.

PROCESSOR STRUCTURE

The figure below, which is taken from Figure 6 of [RAMS79], shows the processor structure for a typical MMBC application. Throughput and fault tolerance considerations necessitate the use of multiple processors for most high-level application processes. Sensor data arrives on 4 busses, and 3 processors on each bus perform the Preprocessing And Bulk Filtering of that data. The Pulse Match Process is handled by 4 processors, the Tracking and Discrimination Process by 3 processors, and the Guidance, Navigation, and Control Process by a single processor. There are two separate bus systems, the sensor busses which carry data from the sensor complex to the MMBC, and a global bus which interconnects all MMBC processors. The processors come in several flavors -- SIMD processors (3 streams each), arithmetically enhanced processors (fast logic for sum of product calculations), and general purpose processors.



The MMBC hardware is modular in nature so that the processor structure can be tailored to the needs of the mission. It includes a microprogrammed general processing element (GPE), an element that combines with the GPE to form a single instruction multiple data stream (SIMD) processor, an element that combines with the GPE to form an arithmetically enhanced processor, memory elements, and bus interface units.

DESIGN STRATEGY

Because techniques for achieving fault tolerance depend on the logical and physical structure of a system, the design process was largely driven by fault tolerance considerations. Also, because the opportunities for fault recovery are different before and after launch, different techniques were chosen for these periods of operation.

Prior to launch, the system maintains a battle ready status by operating in a continuous self-checking mode. Malfunctions are made known to the battle management system responsible for the missile and this leads to manual repair. After launch, manual repair is no longer possible, and fault tolerance is then provided by static masking. Provision is also made for dynamic reconfiguration in case the masking capability is exceeded.

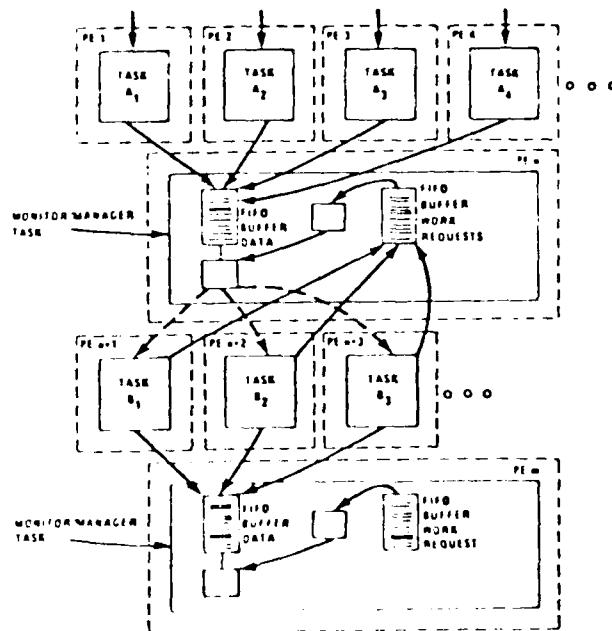
Static masking is implemented in the application tier of the processing model. The basic strategy is described in the following excerpts from [APPL79].

"The architecture we have chosen for the applications software is a pipeline. Tasks at a given level of pipeline must communicate and coordinate with other tasks only at preceding and succeeding levels. So there is no global applications system state, but only communications between successive layers of the pipe. Note that each level of the pipe is a set of identical tasks implementing the same function. The tasks reside in different processors and provide both throughput and fault tolerance. ..."

"Figure 7 shows the PBF function's gross software architecture. The software is in the form of a pipeline; each stage in the pipeline performs some well-defined function (e.g., demultiplexing). ..."

"Each level of the pipeline is supervised by a manager task whose function is to distribute data to the tasks which make up the level. Each level is made up of one or more identical tasks. The number of tasks at a given level determines that level's throughput and fault tolerance. The flow of data through a level is as follows: when a task at that level is idle, it sends a work request to its manager. When the manager receives data from the next higher level, it matches the data with a work request and notifies the idle task. The previously idle task then processes its data, sends the data to the manager on the next lower level

and sends a work request to its own manager."



MMBC Applications Software Pipeline
Figure 7 of [APPL79].

Static masking occurs because of the fact that a processor must request work in order to get work. If the processor dies, it stops requesting work and is automatically out of the job stream. By having more copies of a task than needed to meet throughput requirements, the excess copies can die during the mission without affecting performance. Of course, if too many copies die, performance degrades. If the risk of this occurring is deemed adequately high, a second level of fault tolerance can be implemented using the technique of dynamic reconfiguration, that is, by reconfiguring the processes needed for the remainder of the mission around the processors that are still functioning.

Dynamic reconfiguration is an executive tier function, and the executive was carefully designed to minimize the overhead associated with this task. All processors are tied to a global bus (actually multiple independent global busses to give high performance and fault tolerance) so that processes can communicate no matter which processor supports them. Communication between processes is message based, with the message having a header containing the logical name of the destination process. The communication logic of the originating processor determines if the message destination is to a process which is local to or external to the processor. If external, the message is put onto the global bus, header first, and the processor that holds the destination process recognizes the process name and reads in the message. The result of this arrangement is that the processor sending the message does not have to know the physical location of a process external to it, hence that process can be relocated without having to inform the processor. This eliminates most of the

overhead normally involved in system reconfiguration. For example, the code for a given process could reside in the memory of all processors. If a processor with an active copy of the task died, a surviving processor could be told to activate its copy and the system would continue to function.

3.5.3 TSD METHODOLOGY APPLICABILITY

The vertical structure, process structure, processor structure, and design strategy are as predicted for an interceptor control system. Specification formalisms and design tools were not discussed because of a lack of reference material. Although we do not know the actual thought processes that occurred during the design of the MMBC, the carefully structured result could easily have resulted from the top-down approach described in section 3.4.6. The reasoning is as follows.

The processing requirements of the interceptor mission were well understood and so was the processing load associated with each step. Because the processing load for most steps was too high for a single processor, it had to be distributed over multiple processors. The beginning point of the design process was thus one of deciding on the manner in which the processors were to be organized. This is the point at which the need for fault tolerance begins to direct the design. Most techniques for gaining fault tolerance depend on the inclusion of redundant hardware. The static masking scheme that was chosen was one that simultaneously satisfied the need for extra hardware and the need for a multiprocessor organization.

Note, however, that this scheme has an impact on the communication workload since two messages have to be sent for each task -- a work request message and a work assignment message. The performance requirements for the processes and for the interprocessor links had to be adjusted accordingly.

The design of the application tier identified the number and nature of the processors needed. This was based on throughput, response time, and fault tolerance requirements, and on an assumed order code executing at a rate of one million instructions per second. The hardware assumptions were based on studies of current and near future technology capabilities. It is important to note that even though the nature of the processors was identified (e.g. SIMD, GPE, etc.), this did not bind them to specific hardware, it only constrained the design of the hardware. In a similar manner, the interconnection structure was not bound, only constrained to support the links of the process structure.

The design of the executive tier was also strongly driven by fault tolerance considerations. Two important decisions were made. First, the executive was distributed over the system so that it could survive the failure of any processor. Second, interprocessor communication was bound to a global bus, and bus addressing was made configuration independent, thus making it extremely easy to relocate the processes of a failed processor. Again, these design decisions were not without cost. The adoption of a global bus created a communication bottleneck that required

the design of a high performance bus, and the bus addressing mode required the bus interface units to have intelligence.

The design of the executive level was a cooperative effort involving both software and hardware designers. This led to the implementation of many functions in hardware so as to reduce software overhead and improve performance. The fact that the hostile operating environment required the design of custom hardware was thus exploited to provide an extremely efficient hardware/software mix.

This concludes our discussion of the MMBC. Its design has been presented in enough detail to illustrate that it fits the proposed processing model, that the design strategy of each level is based around techniques for achieving fault-tolerance and throughput, and that the design could have been developed in the top down manner prescribed by the TSD methodology.

REFERENCES

- [ALFO77] Alford, M. W., "A Requirements Engineering Methodology For Real-Time Processing Requirements", IEEE Trans. on Soft. Eng. SE-3, No. 1, pp. 60-69, January 1977.
- [APPL79] Applewhite, H. L., Arnold, R. G., Gorman, T. J., Gouda, M. G., "Modular Missile Borne Computer (MMBC) Software Structure And Implementation", Proc. 1st Internat. Conf. on Distributed Computing Systems Huntsville, Alabama, pp. 725-735, October 1-5, 1979.
- [ARNO79] Arnold, R. G., Ramseyer, R. R., Wing, L. B., Householder, E. A., "MMBC Architecture", Proc. 1st Internat. Conf. on Distributed Computing Systems Huntsville, Alabama, pp. 707-724, October 1-5, 1979.
- [BELL76] Bell, T. E. and Bixler, D. C., A Flow-Oriented Requirements Statement Language, Technical Report TRW-SS-76-02, TRW Defense and Space Systems Group, Redondo Beach, Cal., April 1976.
- [KINN79] Kinney, L. L., Johnson, W. D., Ramseyer, R. R., Stephens, K. L., "Modular Missile Borne Computer Hardware Modules", Proc. 1st Internat. Conf. on Distributed Computing Systems Huntsville, Alabama, pp. 736-746, October 1-5, 1979.
- [McCL75] McClean, R. K., and Press, B., The Flexible Analysis, Simulation, And Test Facility: Diagnostic Emulation Technical Report TRW-SS-75-03, TRW Defense and Space Systems Group, Redondo Beach, Cal., October 1975.
- [RAMS79] Ramseyer, R. R., Arnold, R. G., Applewhite, H. L., Berg, R. O., "The Modular Missile Borne Computer Architecture From The Requirements And Constraints Point Of View: An Overview", Proc. 1st Internat. Conf. on Distributed Computing Systems Huntsville, Alabama, pp. 747-756, October 1-5, 1979.
- [SMIT77] SMITE Reference Manual, RADC-TR-77-364, Rome Air Development Center, Rome, New York, November 1977. AD# A049 038.

3.6 Distributed Data Processing Systems Illustration

Distributed Data Processing Systems take on a variety of forms and sizes. This section presents a broad description of these systems in order to clarify the problem domain, the system solution domain, and the constraints which affect the solution domain. It also describes how the TSD Methodology, presented in Section 3.4, can be applied to select an appropriate design path from the problem domain to the system solution domain. To that end, a specific example is presented, to which the important aspects of the methodology are applied.

3.6.1 Introduction

Systems can be categorized by classifying them within different dimensions of criteria. Here, we select a few of these dimensions to attempt to characterize Data Processing Systems:

- applications vs. support
- environment (existing support to be used as primitives)
- constraints (time, space, cost, etc.)
- system solution (what kind of solutions are expected)

Data Processing applications are typically classified as those in which a large amount of "external", complex data are processed and a relatively small amount of computation is required for each set of data processed. The data are often processed in a transaction type manner (a specific set of output for each specific set of input). This is in contrast to:

- numerical applications
 - Usually, a small amount of data is input or output, but a large amount of numerical calculations are performed.
- control applications
 - Usually, the input and output are very simple control signals, and very little processing is performed.

Unfortunately, there is no clear-cut criteria for pigeonholing specific applications; there is almost always a hazy boundary, and components which have distinctly different characteristics may be present in any specific system. However, a reasonable definition of a Data Processing System is one in which the ratio of data movement to data computation is (in some sense) high.

We will further decompose Data Processing Systems into two classes: application systems and support systems [ALFO79]. Application systems are those in which the semantics (meaning) of the data being processed actually is known by the system. For instance, in a patient monitoring system, the system knows that the data is the output of sensing devices attached to patients. In contrast, a support system is one in which the

semantics of the data being process is not known. For instance, a generalized database system does not know whether the data being processed represents patient's temperatures, automobile prices, or oil well depths.

The environment normally supplies a significant amount of processing support. This support might be as small as a file system, or it may be as large as a full operating system with database and utilities. The design and implementation effort is significantly affected by the level of support available.

The major constraints that are normally applied to Data Processing Systems are:

- Performance

Here, the emphasis is put on obtaining the largest amount of work from the system in a given amount of time (throughput). Load averaging is involved, and there is little intent to allow the exclusive processing of extremely high priority components (as in real-time systems). However, there are often constraints on the response time required by certain requests in an interactive system.

- Maintenance Costs

The total life cycle of the system must be considered in the process of design. The cost involved in maintaining the hardware and software often is an extremely important factor in picking specific system components. The availability of maintenance contracts can significantly affect the system design.

- Expandability and Enhancibility

Most Data Processing Systems expand beyond the scope of the application for which they were designed originally. This may occur in one or more of several dimensions: More may be data maintained than originally expected, there may be greater activity of the system (more users) than expected, or new enhancements may be required to keep the system up to date.

There are several other kinds of constraints that can be applied to Data Processing Systems, but often have little affect on the design. Access security is normally supplied by the executive system in which the Data Processing System is embedded. Reliability is often handled by checkpoints and backups. Real time constraints are seldom applied. Constraints on the physical environment (space, temperature, etc.) usually are not significant factors in design decisions.

The solution systems designed for Data Processing Systems seldom are on the cutting edge of technology. Hardware is usually off the shelf mainframes or mini/microcomputers; customized options are often selected. However, custom discrete logic and/or VLSI are not used. Likely reasons for excluding custom hardware might include increased cost (both initial and maintenance), lack of experienced personnel (training costs), and lack of need (older technology seems to solve the problems). Software components range from off the shelf, through customized packages, to

specially written custom software.

The previous discussion has dealt with general Data Processing Systems; however, here we are interested specifically in Distributed Data Processing Systems. There are four major reasons that a Data Processing System might have a distributed design:

- Geographic Constraints

The data manipulated by a Data Processing System may be obtained from sites which are geographically distant. The system may have to collect specific distributed data, transform these data, and transmit appropriate aspects to specific sites. The physical locations of the sites may be part of the requirements specification.

- Performance Constraints

Due to computational complexity of the application, there may be no single processor capable of producing the performance that is required. Distributing the computations to multiple processors may be the only way of obtaining the required performance.

- Expansion

Although the initial application might be solvable on a single processor system, the desire for future expansion may dictate a distributed design, one that will be easy to expand later.

- Cost

Although an application may have a system solution which uses a large mainframe, the cost of such a system may be prohibitive. A distributed system of mini/microcomputers may have the same computing power but be much less expensive.

3.6.2 Applying the TSD Methodology

The most important tools available to a designer are his own skill, ability, ingenuity, and experience; no methodology can replace these. Instead, the methodology represents a roadmap through a very large, complex catacomb of design decisions through which the designer must traverse. It is intended to give guidance as to when the designer should apply different aspects of his art to specific components of the design problem. It should give insight into when to apply the art, not enhance the art being applied.

The specific techniques used in different steps of the methodology also may be part of the designer's art. They may be dictated by a number of different variables, among which are the type of system being designed, the degree of formality in the specification, the tools available, and the designer's experience. In this section, techniques that may be applicable to Distributed Data Processing Systems are discussed.

The methodology has two major tasks: system architecture design and binding. Although it may seem that (from the specification) these two tasks are strictly sequential and independent, there actually is a very

strong logical interconnection between them. Many of the physical activities described in the binding task may have (and probably have) been logically performed in the system architecture design task. In the refinement of the design, the design decisions are based on 1) constraints and 2) knowledge of what system components exist commercially and the cost associated with building (currently) nonexistent system components. In other words, the design must be a pragmatic one, not a theoretical one. If the components developed during the design cannot be bound to existing or buildable realizations, then the design is faulty. With this in mind, it is clear that the design decisions must be based on concrete assumptions as to how system components eventually will be bound to realizations. Thus, the binding task is mainly intended to:

- bind a few components about which assumptions have not yet been made
- determine the compatibility of previous assumptions (superficial binding) and modify those assumptions to make them compatible
- generate hardware and software requirements for system components not commercially available and state option selection for those that are commercially available

The heart of the System Architecture Design Phase is the action of decomposing the processing model (the first major step of Subsystem-Refinement). The final characteristics of the system are almost completely determined by the decisions made here. The manner in which the decomposition takes place significantly affects the quality of the final design and the speed with which the design is completed. Thus, it is important to understand the factors upon which these decisions are based and have some specific technique for performing the decomposition.

The major factors affecting the decomposition are:

- knowledge of the application
- knowledge of the applicable algorithms
- knowledge of the performance of hardware and software that is available or can be built
- recognition of the constraints

All of these factors must be kept in mind as the designer considers different decomposition options. The loop (iteration1) after process/processor allocations is a formal verification of how adept the designer is at juggling these factors during the decomposition.

Two well-known techniques for performing the decomposition are functional decomposition [BERG81] and data flow analysis [PAGE80]. Since these are well-known, they will not be discussed further. A more interesting question relates to the selection of those processes/processors that are considered as candidates for decomposition.

The canonical problem is the one in which a set of n processes are allocated to a single saturated (virtual) processor. (A saturated processor is one which cannot be bound to a real machine and meet the performance requirements of the processes allocated to it.) There are three major alternatives: 1) decompose the processor into m processors and reallocate the n processes in such a way that performance requirements can be met, 2) deallocate the "appropriate" number of processes from the processor (so that the performance requirements of the remaining processes can be met by the processor) and reallocate them to other processors, and 3) decompose certain of the n processes and reallocate specific resulting components to other processors (so that the performance requirements of the remaining components can be met by the processor).

Option 1 should be used as a last resort, only after options 2 and 3 have failed. It introduces additional processors which a) increases the cost of the ultimate system, b) increases communication cost, and c) increases the complexity of the system.

Option 2 is applicable if there are unsaturated processors available and the reallocation of the processes does not saturate them. This is the preferred option, but random reallocations can unnecessarily increase the number of communication links required between the processors.

The processor topology is inherited from that of the processes; given a specific process topology, certain processor-processor communication links may be required because of the associated process-process communication links. As processes migrate to different processors, the processor-processor communication topology may have to be modified to reflect the new allocation. One solution is to simply accept this situation and pay the price of the increased communication. Another is to introduce "packet transmitting" processes that shuttle information from processor to processor.

Option 3 may be applicable when there are unsaturated processors available, but reallocation of entire processes saturate them. In this case, reallocation of subprocesses (process components produced by the decomposition) may not saturate the unsaturated processors. This option also tends to reduce the processor communication problem mentioned above.

The overall objective is to produce a system (which meets all constraints) with the fewest processors and fewest communication links between them. Option 1 increases both the number of processors and the number of links; options 2 and 3 potentially increase the number of links.

The problem is very similar to grouping and assignment problems in graph theory [BOND76, EVEN79]. How do you group the nodes of a graph in such a way that the connecting arcs between groups is minimized and certain constraints on the nodes within each group are met? Research into graph theory may yield appropriate algorithms for helping the designer to select candidate decompositions.

Determining performance capabilities of different components (both hardware and software) of the system is another extremely important aspect of the TSD Methodology. Benchmarks may be useful for gross estimation of general performance capabilities and comparisons between different hardware. However, extreme caution must be taken in selecting the type of application upon which the benchmark is based. If the test application does not have computations similar to the problem application, essentially no inference can be drawn.

Analytical models, such as queuing theory models may be applicable for determining the order of an algorithm and its appropriateness in an application. Simulation with languages, such as GPSS, may also provide insight into the ultimate performance of a specific system design.

Often the performance of a subsystem is dependent on the performance of a critical section of code, and the total performance can be calculated in a straightforward way once the performance of that code is known. Thus, an excellent alternative is to program the critical section of code and do performance measurements.

Emulation and simulation are not particularly applicable to determining performance capabilities in Data Processing Systems.

3.6.3 An Example - Digital Land Mass System (DLMS)

The informal specifications of a cartographic database [NAGY79] are presented in this section. This will be used as a basis for presenting the general properties of the TSD Methodology and how it is used dynamically. The example presented here is very simple. It has been constructed in order to demonstrate the use of the Methodology and not to demonstrate the work required to develop the detailed design of a complex system. We will refer to this example as the Digital Land Mass System (DLMS).

DLMS consists of a set of data (the database) and a set of transactions. The system is capable of accepting, retaining and answering questions about three different kinds of objects embedded in an absolute latitude-longitude coordinate system: points, lines, and areas.

Each data group has the same conceptual form, a 3-tuple: <feature, type, coordinates>. The feature component specifies a unique name of the object (e.g., Missouri River, Pike's Peak, Lake Michigan, etc.) within a specific type. No two objects in the database of the same type can have the same feature. The type component specifies the kind of object being described. In this database, there are only three kinds of objects: points, lines, and areas. The coordinates component specifies a list of 2-tuples which are interpreted as latitude-longitude coordinates. If the type of the object is point, the coordinate list corresponds to a sequence (possibly only one) of points which are conceptually associated (such as the positions of telephone poles along a telephone line). If the object is a line, the list is of arbitrary finite length; the sequence of points defined by the coordinates are conceptually joined (in sequential order) by straight line segments in order to approximate the actual line being

defined. If the object is an area, the list is of arbitrary finite length; the sequence of points, joined by straight line segments, represents a closed curve enclosing the area being defined. The database consists of a specific set of objects defined in the above manner.

There are four primitive transactions: create, find, update, and plot. Each command is input to the system along with a number of arguments. In each case, if an argument is of the wrong format or structure, the database remains unaltered and an error message is given.

The create command has three arguments: a feature, a type, and coordinates. If any object currently in the database has a feature and type identical to that specified, an error message is returned and the database is not altered. Otherwise, an object with the given feature, type, and coordinates is added to the database.

The find command has two arguments. The first is either empty (null) or a feature; the second is either empty or a type. Only one of the arguments may be empty. If only a feature has been supplied, then those objects (a maximum of three) named with that feature are returned as output data (if such objects exist in the database). If only a type has been supplied, then all objects with that type are returned as output data (if there are any such objects). If both feature and type are supplied, then the object with the specified feature and type is returned as output data (if it exists). The database itself is not altered by this command.

The update command has three arguments: a feature, a type, and coordinates. If no object in the database has the feature and type specified, an error message is returned and the database is not altered. If there is an object in the database with the feature and type specified, then one of two actions occurs: if the coordinates are empty, then the object is removed from the database; otherwise, the coordinates of the named database object are replaced by those specified in the command.

The plot command has four arguments: each is a latitude or longitude coordinate. Collectively, they define a rectangular area of interest. The plot command returns as output the set of objects (or portions thereof) that lie within this area. The database is not altered by this command.

This chartographic database system is intended as a subcomponent of a larger system which must be able to manipulate geographic data. Thus, it is not intended for direct use by an end user; there is some software interface that transforms the users' input into the commands specified above and interprets the output and displays it to the user in an appropriate manner. Thus, there is no question about how error messages or output responses are displayed to the user or how the response from the plot command actually is plotted.

The intended size of the database is between 10^{15} and 10^{16} objects. Thus, a significant amount of on-line secondary storage will be required. It also is anticipated that the database may be extended beyond this initial volume.

The system is intended to be used in an interactive environment. Thus, although this is not a real-time system, there are certain requirements on the response time required to execute commands (as opposed to a batch oriented system where average throughput might be more important). It is intended that both data entry and query of the database are to be done interactively; thus, techniques that balance data insert and data search are most appropriate. (This is in contrast to techniques that produce very fast data entry but very slow data retrieval, or visa versa.)

Although concrete performance constraints would be required in order to do an actual design, numerical constraints are not presented here because in this exposition, the algorithms used will not be analyzed to a low enough level to be able to apply actual numerical constraints. However, during the exposition, assumptions will be made about the comparison of the numerical constraints and the algorithm analyses.

3.6.4 A Design - DLMS

In this section, the TSD Methodology is applied to the example described in the previous section. The intent is to show the sequencing of the steps involved in applying the Methodology and how results from analyses affect decision points of the Methodology. The intent is not to show the details of the analysis or exactly how the decisions are made.

A preliminary analysis of the system leads to some conclusions about the solution domain. Some form of random access secondary storage is required; this rules out tape. Because of the amount of data to be stored, performance constraints, and reliability considerations, floppy disk is ruled out. This leaves hard disk and bubble memories. Because of the current state of the art for bubble memories and personnel expertise, bubble memories are ruled out, and some form of hard disk is selected for secondary storage.

Because of performance constraints, cost constraints, current technology and the desire to be able to expand later, a distributed architecture seems to be appropriate. Some form of mini/microcomputer is deemed appropriate; however, some software support is required, at least a file system and the availability of a high-level language. This leads the designers to place machines like the PDP-11 and MC68000 in the design space (binding options).

Thus, as the designers start into the design, they have some idea of what hardware and software might be appropriate. They have not decided what hardware and software to use, but they have restricted their search space.

An important aspect to consider before getting too deep into the design is what fundamental approach (top-level idea) will be used to perform the database searches. There are many such approaches: binary search, binary trees, AVL-trees, hashing, B-trees [AH074, HOR076], etc. In fact, the initial technique selected may not be the final technique used; however, it is useful to have a strawman against which to evaluate

design decisions.

Because of the large and unknown amount of data, many of the standard search techniques are inappropriate. B-trees [COME79, HANS81] seem to be applicable to this application. They can handle very large data sets of initially unknown size; they supply relatively good performance; and they can be "tuned" to the properties of the specific secondary storage system. Let us assume that although other approaches initially may have been considered (and some design done based on that choice), m-way B-trees constitute the final selection.

An m-way B-tree is a search tree in which:

- 1) all leaf nodes are at the same level (distance from the root node)
- 2) the root node has at least 2 children
- 3) all nodes other than the root and leaf nodes have between m and $m/2$ (rounded up) children

In such trees, the raw records (or pointers to them) are held only at the leaf nodes. Internal nodes contain search information about the key (of the records) and allow algorithms to select the appropriate child to be searched. The leaf nodes are sorted from left to right, and the B-tree structure supplies a fast access structure for finding the appropriate entry (or where it would be if it were present). The search, insert, and delete algorithms are straightforward, and the height of the tree determines the number of nodes that must be interrogated in order to find the appropriate entry. Since each node must have at least $m/2$ children, it can be shown that this does not exceed $\log\{(N+1)/2\} + 1$. (Here, the logarithm is taken base $m/2$, and N is the number of leaf nodes.) For instance, if $m=20$ and the number of entries does not exceed $2 \times 10^{16} - 2$, then no more than 7 nodes must be interrogated. (m usually is determined on the basis of the key size and the disk block size.)

The initial process structure is shown in Figure 3.2(a). Here two independent user processes (U_1 and U_2), running on their own processors, interact with the DLMS process; these are outside the scope of our design. The DLMS executes on a single virtual processor. Because of the transaction nature of the specifications, a natural first level decomposition is to recognize the four different commands by introducing processes to handle each of them (see Figure 3.2(b)).

Upon further analysis of the requirements, it becomes clear that the database will have to be searched in several different ways -- the find command searches on feature and type, and the plot command searches on coordinates. It is decided that separate access structures should be built for each type of search to insure maximum performance for that specific kind of search.

One possibility is to have a different search structure for each component type: feature, type, and coordinate. However, there are only three different values for type. The following compromise is chosen:

There is one access structure for coordinates and three access structures for feature, each containing types of only point, line and area, respectively. Given this architecture, if a type is specified in a find command, only the corresponding access structure need be searched. If only a feature is specified in the find command, then three B-trees must be searched.

Clearly Plot is the critical process because a spatial search must be done in the rectangular area of interest. It is selected as the critical component to analyze and decompose. Two basic functions must be performed: (1) determine the objects that intersect the area of interest, and (2) extract the portions of those objects that lie within the area (see Figure 3.2(c); for brevity, only the refinement is shown; SIR stands for Search/Insert/Retrieve). We will assume that (2) is not difficult once the object is identified. Thus, (1) becomes the critical process of interest.

One way to perform this search is to: (1) identify all objects lying within the appropriate range of x-coordinates (corresponding to the sides of the rectangular area of interest), (2) identify all objects lying within the appropriate range of y-coordinates (top and bottom of the area of interest), and (3) take the intersection. If a separate access structure were held for the x coordinates (that define the objects) and the y coordinates, then (1) and (2) above have simple algorithms and potentially can be done in parallel. Thus, the decision is made to refine the previously conceived single access structure for coordinates into two distinct ones -- one for the x coordinates and one for the y coordinates (see Figure 3.2(d)).

Now that a sufficient amount of design has been done to crystallize a probable structure for a critical area of the system, it becomes evident what the design of the remaining portion of the system should be (see Figure 3.3). As an example, consider the create command. Upon receiving an object, the type is known so the appropriate feature B-tree can be searched. If there is no object of the given type and name, Create puts the primitive object in the database (getting a pointer back) and puts the pointer in the appropriate place in the appropriate feature B-tree. It then performs a similar action for each of the x and y coordinates in the list of coordinates. Update is similar to Create; Find only does searches; and Plot has already been discussed.

The above scenario corresponds to recursively traversing the procedure Subsystem-Refinement to the fourth level ($j=4$) while performing some incidental backtracking. Note that no mention has been made of refining the processor structure or which of these processes are to be performed in parallel on separate processors (so far, there is just one processor). Subsequent levels of refinement will address these questions and potentially refine the process structure.

Supervisor, Create, Find, Update, and Plot all have simple, similar supervisory activities that do not require much execution time. Because their functions are similar, they may have common utilities; thus, it is potentially advantageous to place them on the same processor. The five SIR processes are very similar in nature, but their processing load is

anticipated to be drastically different. The three feature SIR processes are accessed by Create, Find and Update. These normally perform one probe into the B-tree and do minor manipulation of the local region found in the search. However, the two coordinate SIR processes are used by Plot to extract large amounts of data; two probes are made and all the data "in between" is returned. Thus, an appropriate processor structure might be to assign the three feature SIRs to one processor and each of SIR X-coord and SIR Y-coord to two other processors. Although the algorithm in Intersect and Extract Portion are simple, they will be performed on a large amount of data. They can be assigned to their own processor. This suggests a processor structure of that shown in Figure 3.4. Note that the majority of the processing power (three processors) is allocated to the critical activities of Plot. Also note that the structure shown has the minimal communication (six links) required by a connected system of seven processors.

After consistency, performance, and inference analysis, this processing structure seems appropriate and we can consider the processor structure to be completely refined. Thus, we start on the subsystem design (recurring into Subsystem-Design at $i=2$). The SIR processes can be viewed as accessing a virtual B-tree machine; there are specific instructions to search, insert, and retrieve information from a B-tree (the B-tree conceptually residing on disk). At this stage, the B-tree machine must be designed. (Of course, all the five B-tree machines have the same design.) Although such a B-tree machine could be built completely in hardware, a more standard approach (and the one we take here) is a combination of hardware and software (processors and processes).

The basic structure of the processes of the B-tree machine is shown in Figure 3.5. (Here we assume a single processor consistent with prior decisions, although we could refine the processor structure more.) SIR* represents the interface between the specific SIR process making the request and the B-tree machine. Decode interprets the request and dispatches the appropriate action. Each of Search, Insert, and Retrieve execute sequences of commands on the processor to achieve the required function.

Let us assume that the designers believe that this is the design that they want, but they are still not sure of the performance characteristics. As an inference tool, they implement certain portions of the system and execute against what they believe is a representative set of queries. Unfortunately, they find that the performance is below requirements. After analyzing the detailed execution results, they find that the poor performance is due to excessive I/O to the disk. However, on further analysis, they find that they had not recognized the principal of locality (if a certain portion of the B-tree has just been searched, it is likely to be searched again in the near future) and that much of the I/O can be eliminated by holding the most recently accessed nodes of the B-tree in primary storage buffers.

Thus, the designers recurse one more level into Subsystem-Design ($i=3$) by assuming that Search, Insert, and Retrieve have available to them a virtual disk machine (see Figure 3.6). Here, the starred processes represent interfaces to processes which need access to the virtual disk

machine. When requests for access to the disk are made, buffers which hold recently accessed information are searched first to see if the information is available in primary storage; if it is not, the disk is accessed and the buffers are updated.

Now assume that after modifying the previous implementation to reflect the virtual disk machine subsystem, performance seems to be within bounds. Although it cannot be assumed that the final system will work within the required performance bounds, the designers now have a much clearer understanding of the overall structure of the system and may have already identified places where the system can be optimized if later necessary.

The preceding scenario has been intended to show the general overall nature of the TSD Methodology. There are many aspects of the design which have not been addressed. For instance, what kind of interfaces between processes and processors were assumed and how is the disk system organized? Are all the B-trees on one disk (causing seek thrashing), or are they segregated? The Methodology has lead to a design that allow these questions to be resolved in multiple (acceptable) ways. It has not force the designers to "paint themselves into a corner".

What will happen as the system grows? There are several ways in which the system might expand: more data may be held, more users may be put onto the system, or more types of objects may be added. The current design allows graceful growth in all these directions. As the number of objects grows, the disk system can be expanded and performance can be enhanced in several different ways depending the location of the bottleneck. For instance, Extract Portion and Intersect can be put on separate machines; the three feature SIR processes can be put on separate machines; SIR X-coord and SIR Y-coord can be refined by dividing the coordinate space into fixed regions and assigning different processes (and processors) to each region. As the number of users grows, the processes on PS processor can be split up and put on their own machines (along with the modifications above). As the number of types grows (so that user defined types are available and a finite by unbounded number of type are possible), the three feature SIR processes can be logically combined and then split into two processes (each on its own processor): one searching on feature and one searching on type.

In summary, the Methodology has produced a design that is flexible to growth and enhancement in many different dimensions. For instance, the basic design is applicable to different access structures other than B-trees, different commercially available machines and different disk systems. Performance and cost are the two driving forces that affect the implementability of the final system.

3.6.5 Conclusions

The TSD Methodology has some interesting and unique properties. Many of these are important to computer system design in general, but a few are particularly important to Data Processing System design. For instance,

the Methodology formally recognizes the concepts of support, subsystem and virtual machine. This is extremely important in Data Processing where systems often are built of pre-existing building blocks. This not only allows the designer to use already available components when applicable, but is a tremendously powerful conceptual tool for reducing apparent complexity when new components must be designed and built.

The Methodology is flexible in that it allows the designer to examine design paths, select those of critical importance, and use the design developed along these paths to mold the remainder of the system. The designer can move about the design space in a flexible and yet structured way. However, it is rigid in that it continually reinforces that the constraints are a fundamental criteria against which the final design must be judged. At each decomposition point, the designer must relate the ramifications of his decisions to the constraints. Although the designer can suppress considering the constraints at certain points, it is still brought to his attention and he must formally address that he is suppressing such considerations.

The Methodology formally recognizes that either the process structure or the processor structure may be the driving factor behind the design. However, it does not force the designer into excluding one aspect for the other; in successive levels of decomposition, the designer can switch from one viewpoint to the other.

Formal specification languages and checks are fostered by the TSD Methodology, and many of the formalisms present in other methodologies [ALFO79] are applicable here. Our Methodology also distributes verification checks to different decision points. The intent is to insure a sufficient degree of verification without forcing the designer to spend excessive time. The Methodology also formally recognizes the environment that interfaces with the system to be designed.

Although this section has only applied the Methodology to an application Data Processing System, the Methodology can be applied in a similar way to support systems. For instance, the techniques used in the design of the DLMS may be extendable to a generalized database system (in which the semantics of the data is not known). In support systems, the constraints may be less well defined because the use of the system and the semantics of the data are not known. In fact, processes may be created at execution time that cannot be analyzed at design time. Here, queuing theory models may become a very important analytical tool.

REFERENCES

- [AH074] Aho, Alfred V., Hopcroft, John E. and Ullman, Jeffrey D., The Design and Analysis of Computer Algorithms, Addison-Wesley, 1974.
- [ALF079] Alford, M., "Requirements for Distributed Data Processing Design," Proc. 1st Int. Conf. Distributed Computing Systems, pp. 1-14, Huntsville, Alabama, October 1979.
- [BERG81] Bergland, G. D., "A Guided Tour of Program Design Methodologies," Computer, pp. 13-37, October 1981.
- [BOND76] Bondy, J. A. and Murty, U. S. R., Graph Theory with Applications, North Holland, 1976.
- [COME79] Comer, D., "The Ubiquitous B-tree," Computing Surveys (11,2), pp. 121-137.
- [EVEN79] Even, Shimon, Graph Algorithms, Computer Science Press, 1979.
- [HANS81] Hansen, Wilfred J., "A Cost Model for the Internal Organization of the B+-tree Nodes," TOPLAS (3,4), pp. 508-532, October 1981.
- [HOR076] Horowitz, Ellis and Sahni, Sartaj, Fundamentals of Data Structures, Computer Science Press, 1976.
- [NAGY79] Nagy, George and Wagle, Sharad, "Geographic Data Processing," Computing Surveys (11,2), pp. 139-181, June 1979.
- [PAGE80] Page-Jones, Meilir, The Practical Guide to Structured Systems Design, Yourdon Press, 1980.

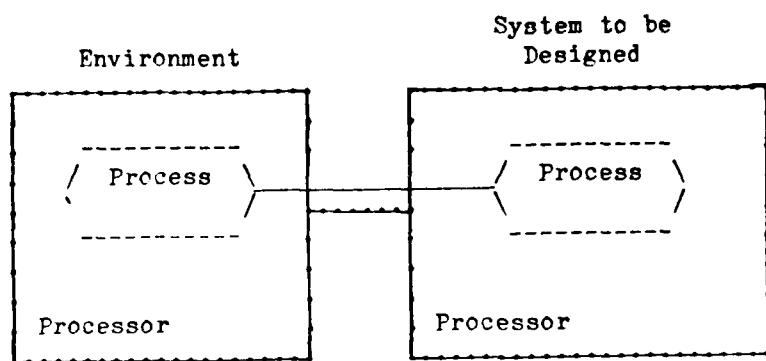
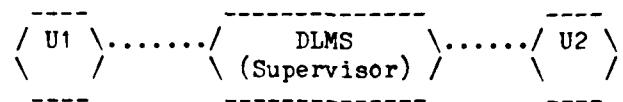
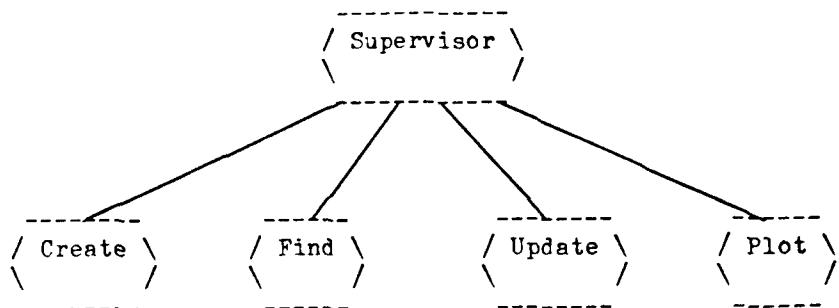


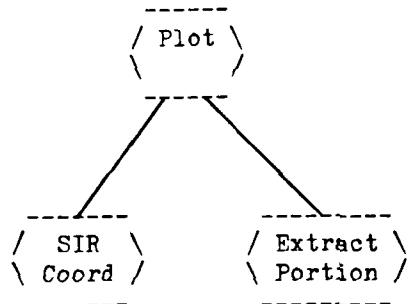
Figure 3.1: Initial Processing Structure



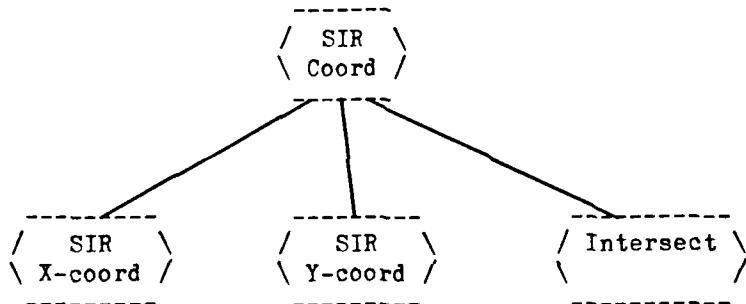
(a) Initial Process Structure (j=1)



(b) Initial Process Refinement (j=2)



(c) Further Refinement (j=3)



(d) More Refinement (j=4)

Figure 3.2: Process Structure Refinement (i=1)

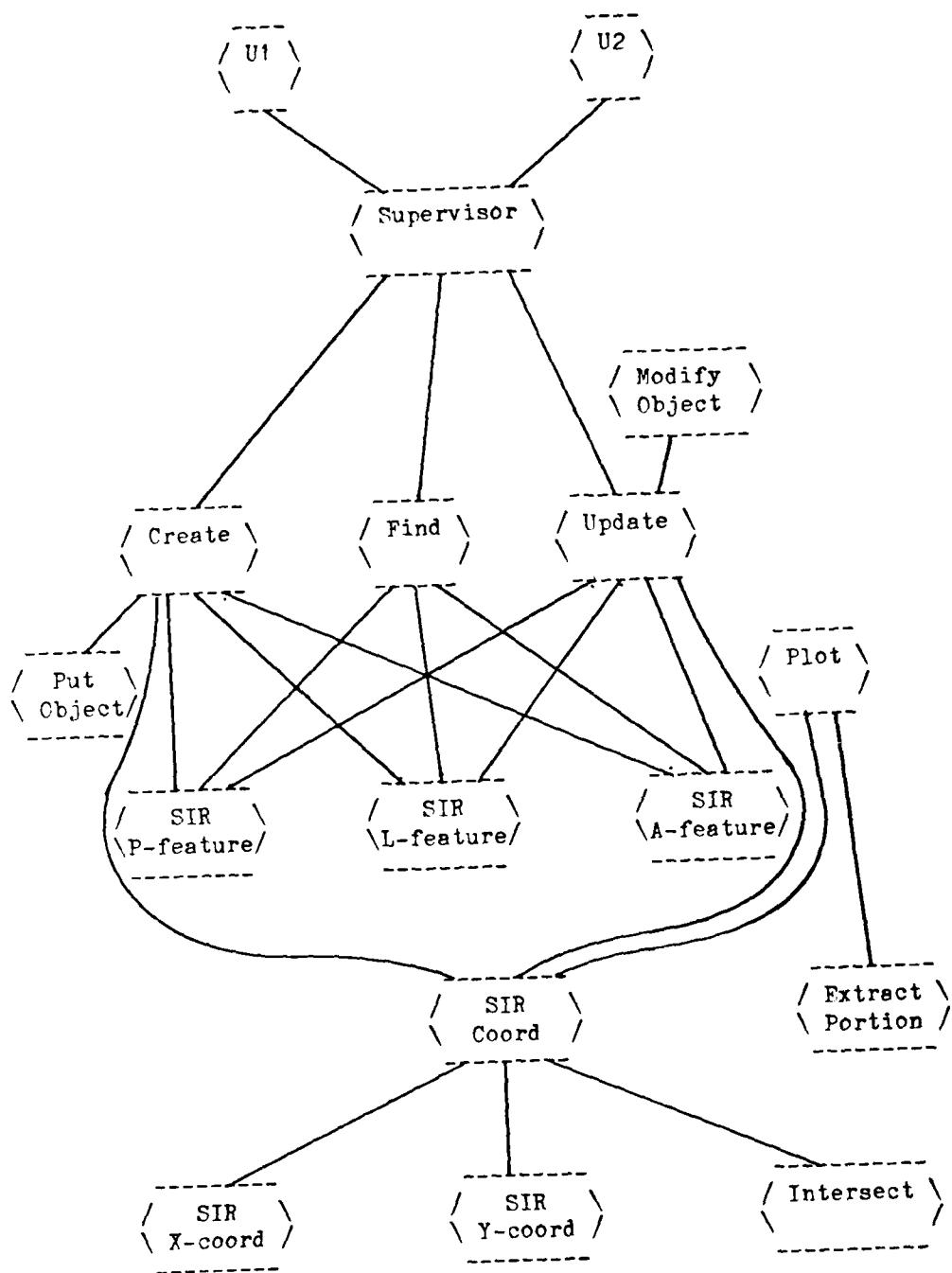


Figure 3.3: A Process Design ($i=1$)

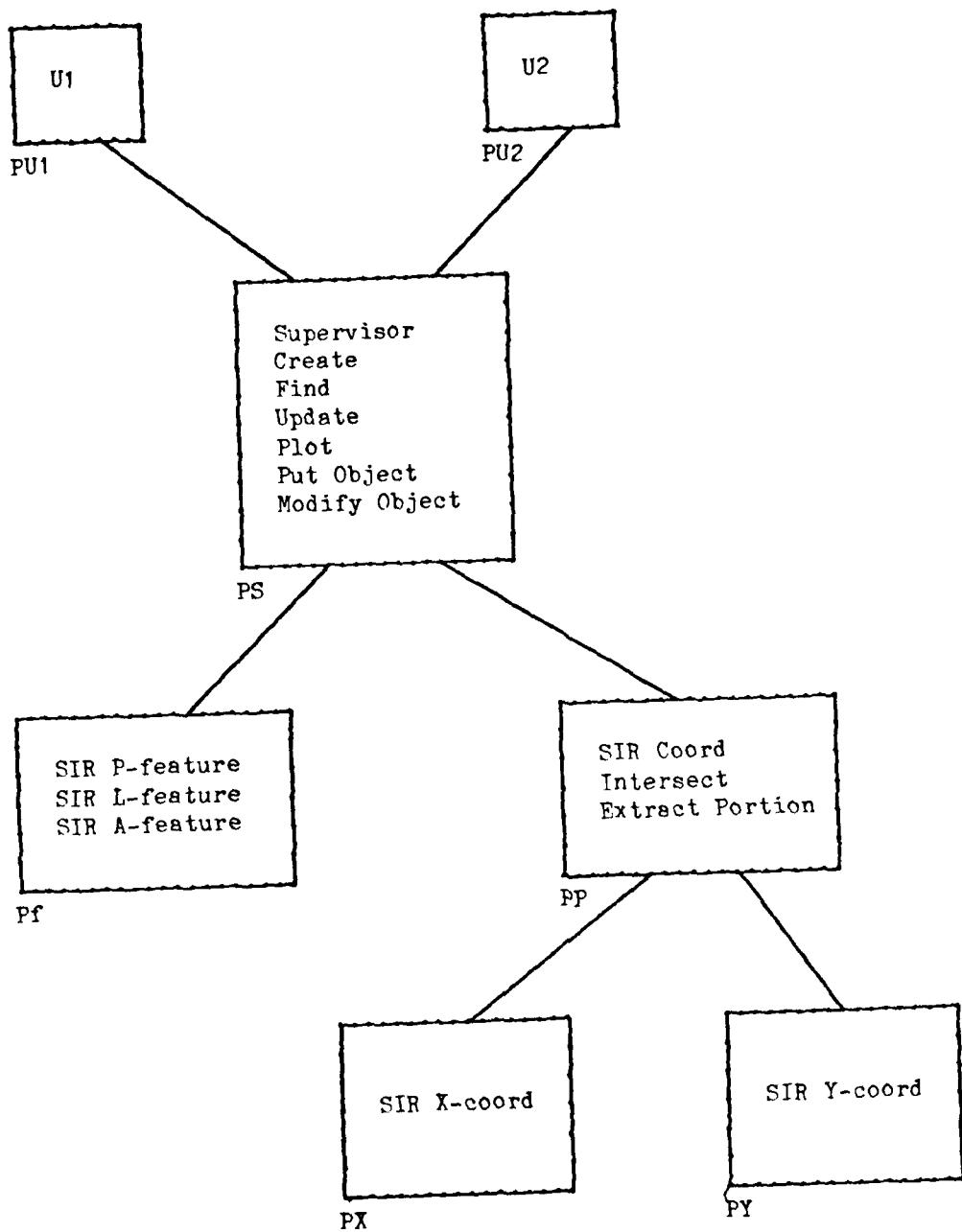


Figure 3.4: Processor Structure and Allocation (i=1)

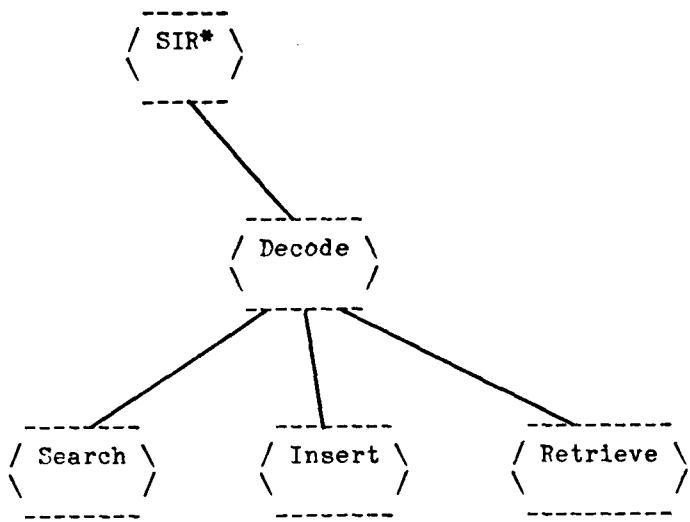


Figure 3.5: B-tree Virtual Machine (i=2)

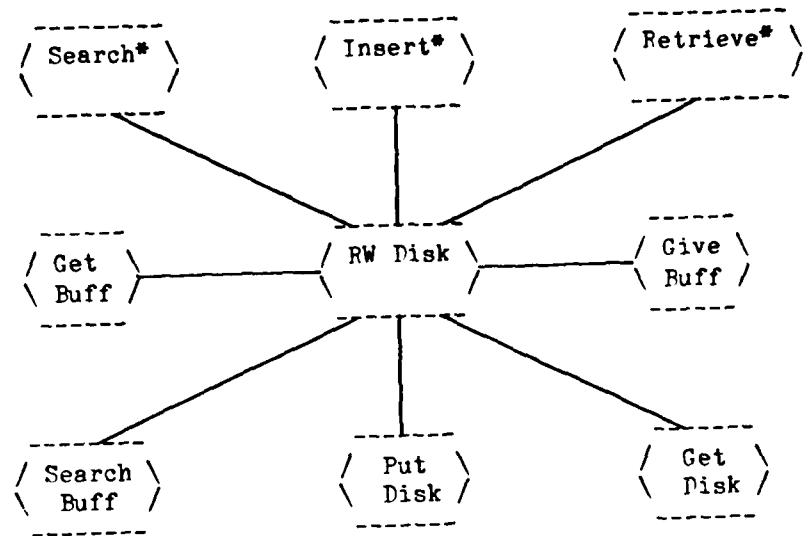


Figure 3.6: Disk Virtual Machine (i=3)

4. TSD FACILITY DEVELOPMENT MASTER PLAN

4.1 INTRODUCTION

System designers and programmers have long felt the need for computer aided assistance in performing their jobs. Software tools represent one of the mechanisms by which the appropriate computer services can be provided. Many of these tools have evolved since the days when the first programs were written, since there always has been the desire to let the computer do more of the mechanical work required to define, design, build, and maintain any computer based system. The justification for this has been primarily economic: people will be more productive in doing any task if they have the correct tools to help them with their work. As an added benefit, it is also usually true that the resulting systems will be more reliable. The definition of what is a "correct" tool, however, varies with the project, the personnel, and the times. Of course, if any new support system is not easy and natural to use, there will be a definite tendency to avoid using it and to continue with the older more familiar methods. Thus what is needed is a comprehensive computerized support system that will provide a set of the appropriate tools to a group of users in a friendly, convenient, easily learned fashion.

GOALS

The goal of this work is the creation of a computer based system that will support, in a cost-effective manner, a computer oriented project from conception through performance evaluation. This includes enhancements in the productivity and the quality of system design efforts within DoD through the use of systematic design approaches. The resulting methodologies are to be supported by the large-scale highly-integrated set of computer aided design and management tools that compose the system. The implementation of this system should take advantage of as much existing technology as possible in order to become operative in as short a time as possible. However, this emphasis on short term utility should be balanced against the long-term need for the system to accommodate new technologies, tools, and methodologies, or extensions to cover the entire system life cycle.

OBJECTIVES

In order to achieve the goals, the following specific tasks were established as the initial objectives:

-- Objective 1:

Develop a conceptual model for an integrated set of design tools to support the first three stages of the TSD Framework (Problem Definition, System Design, Software Design). That is, characterize the computer based environment in which the users will work.

We shall use the expression Total System Design (TSD) Concept to represent the abstract process by which a computer-based system is defined, designed and developed, plus the management of these activities. The TSD Concept establishes the requirements that must be met by the proposed computer aided design system, and hence must be defined before design may begin. A Total System Design Methodology is a particular instantiation of the TSD Concept for one application area.

Usually a system designer will have available a collection of computer based tools (such as a text editor, a language compiler, etc), but this collection does not necessarily form an environment. An environment is the set of services provided to a user when a collection of tools are integrated together to form a cohesive set. We shall call the set of services provided by an integrated set of tools supporting a TSD Methodology a Total System Design Environment. Thus the TSD Environment forms a conceptual model of a group of services that support the first three stages in the life cycle of a project, with the support following a set of guidelines (from the appropriate methodology) to increase user productivity and system reliability.

This task ensures that the functionalities and interfaces required by the TSD Methodologies are defined and included within the TSD Environment. This task also will enforce the integration of the various tools that cover the appropriate system life cycle stages, and hence will demonstrate the ability of the TSD Environment to support all appropriate TSD Methodologies.

-- Objective 2:

Investigate design alternatives for the TSD Environment, and select a specific direction to elaborate. Apply the selected approach to develop a high level design proposal for a TSD System prototype.

We shall call a computer implementation of a TSD Environment a Total System Design System. When a TSD System is installed in a particular computer center, unique features at that center may have to be accommodated within the TSD System itself. Special emulation facilities, unusual applications, or customized tools may all represent factors that may cause the basic TSD System to vary from one installation to another. These variations, however, represent local enhancements of the System, not basic changes. The collection of all of these possible enhanced versions of the System will be called the TSD System Family.

A general design must be found for TSD Systems that incorporates all of the necessary factors identified in the TSD Environment, and also allows for the open-ended inclusion of new technology. Various approaches need to be evaluated to produce the most cost-effective final design proposal. The creation of a specific design for a prototype system allows the general design approach to be evaluated.

The proposed TSD System design is called a prototype because it will be the first one to be implemented. It is assumed that lessons learned from implementing and using the prototype will result in modifications

that produce the final production TSD System design. Depending on the extent of the changes required, the final system may evolve from the prototype or it may come from a complete restart of the design process.

- Objective 3:

Develop a phased implementation plan for a TSD System prototype that emphasizes both the immediate productive use of existing installations and the long-term research and development role of the System.

The TSD System prototype design proposal must be implemented initially by using currently available technology in order to obtain a running prototype system within a reasonable budget of time and effort. The implementation plan must be in phases to allow the immediate exercising of parts of the overall system as they become useable. This approach increases the short term utility of the effort, and at the same time provides for critically important user feedback. The choices made during the development of the TSD System prototype design proposal must be evaluated within the context of an actual project effort, and so the plan must be able to accommodate revisions based on such information. The evaluation must be done in terms of the computer aided design and productivity enhancement goals previously stated.

The TSD System prototype, when actually implemented, will also serve as an excellent test vehicle for the research and development necessary to create new tools and methodologies. These new items may replace or add to existing items, or they may serve to expand the system life cycle coverage. Thus the implementation plan must stress flexibility to allow the research results to feed back immediately into the System for production use. The System implementation plan must also stress portability to allow the results to be distributed to other installations for testing and production.

Facility Considerations

A computer installation that is running the TSD System Software, and is providing all of the necessary support material and personnel, will be called a Total System Design Facility. The essence of the TSD Facility is in its support of one member of the TSD System Family, and thus we may have the TSD Environment implemented at many different installations.

Since each TSD Facility may have something unique to offer (such as special hardware or special tools), a user physically located at one TSD Facility site would have local access to only the one member of the TSD System Family that was adapted to the local capabilities. As a consequence, each TSD System Family member program must have the ability to communicate with all other Family members in order to provide each user with the complete TSD System capabilities (i.e. the union of the capabilities of all of the TSD System Family members).

The core of the TSD System Family is defined as the intersection of the capabilities of all of the potential TSD System Family members. In other words, the core represents the features that are available locally

at all of the TSD Facilities. These features are the computer based functions that are implemented uniformly for all TSD Methodologies, independent of whatever application area is being considered.

APPROACH

Objective 1

The first objective will be achieved by developing a TSD Environment based on the following basic principles:

-- Capture the nature of project organization

Most projects are organized in a hierarchical or matrix fashion. This provides for both supervision and grouping of functions within the project. In a similar manner, the computer environment seen by a user should reflect his level of function within the project organization. This may be readily achieved by organizing the TSD Environment itself into a corresponding hierarchical or matrix structure.

A user working in any given environment should always be able to define a new sub-environment that has access to a defined set of tools and project information. The user should be able to control access rights to any sub-environment that he has created. This will also help satisfy the need for specialization, allowing a user to define an environment tailored to his own special requirements.

-- Complete documentation and configuration control

In order to maintain consistency and control, all information about a developing project must be stored in one data depository. Users may have local copies of pertinent data for their own use, but changes may be made in the central project database only under strict control. This will also support the mechanisms for tracing changes, tracking version numbers, and for maintaining authorization controls.

-- Standardized tool access and control

The TSD Environment will define a set of tools and interfaces that are common for all TSD Systems. This core will represent the components used to build all of the operating sub-environments. In order for this collection of components to form a true environment, the collection must form a cohesive set. This means that each tool may potentially have to accept as input the output of other tools, and all of the tools must interface smoothly with both the project database and the user. Of course, users must be able to add (easily) private tools and interfaces to customize an environment for their own work habits and personal use, and so the model must also provide for these dynamic possibilities.

-- Separation of concerns

Certain major tasks are common for almost all projects, but each project contains tasks of widely varying concerns. System designers and the project manager have quite different requirements for computer assistance in their jobs. In order to aid in the human engineering aspects of using a complex and extensive system, such as the TSD System Family, sub-environments will be predefined that aid particular types of tasks. Current planning calls for three distinctly tailored types: a management environment, a design environment, and a production environment.

Objective 2

The design alternatives for a TSD System that implements the TSD Environment will be developed from the following principles and assumptions:

-- Cooperating installations

The size of this undertaking suggests that there should be multiple simultaneous developments occurring. This will require a strong emphasis on portability (to allow programs and results to be freely interchanged) and for inter-facilities communication (for the rapid exchange of information). A design direction leading to cooperating independent stand-alone system modules should be emphasized.

-- Specialized facilities

It is expected that different installations of the TSD Facility will offer different specialized capabilities. For example, one installation may have a hardware emulation capability that is unique. It may not be cost-effective to duplicate these capabilities in all installations, but workers at another TSD Facility may require access to such unique facilities. Thus we find that the need for portability and communication capabilities mentioned above is re-enforced. However, the design must incorporate such flexibility and still maintain a reasonable efficiency.

-- Workstations

It is assumed that users will communicate with the TSD Facility through local workstations. However, the inter-relationships among the user needs and desires, the workstation capabilities, and the TSD System tool functions, must be carefully explored to insure maximizing the effectiveness of all components. The potential effect of technology (such as graphics, vocal I/O, touch screens, etc) needs to be evaluated.

-- Standardized interface to database management systems

In order to make maximum use of existing technology, it is expected that current database management systems (DBMS) will be used to handle the project database. This expectation must be verified, of course, but the cost of developing a specialized DBMS is large. The disadvantages of excessive generality, size, and slow performance of an existing DBMS must be balanced against the advantages of flexibility and short-term utility. In addition, the use of an existing system allows the core tool set to

interact with the project database through a well-defined, standardized, machine-independent interface (from the tools' point of view). Implementations for a new machine installation would then require mapping the standardized interface, as seen by the tools, into a DBMS that already exists on the new machine. Thus all necessary machine dependencies can be encapsulated. The design characteristics should lead to the most universal and easily adapted interface definition and so must be carefully considered.

-- Standardized core command language and tool set

Command languages, interfaces, and functionalities defined to be in the TSD System core will not change from one installation to another, or from one project to another. This supports portability of tools and applications, plus it allows the personnel to move between projects or installations with minimum retraining. However, the definition and design of such a common core is a significant undertaking in itself. It depends on extracting the essential features from the TSD Environment, and on establishing all of the potential tool/database/human interactions.

Objective 3

The third objective, to create a phased implementation plan for a TSD System Prototype, will be achieved following these guidelines:

-- Evaluate available technology

A demonstration facility should be selected first, since the constraints on the design of the prototype system will depend partly on where it is to be implemented. The existing hardware factors (speed, size, terminal availability, emulation capability, etc) and the existing software tools (database management systems, language compilers, text processors, etc) need to be evaluated and factored into the design proposal. This should result in an identification and partial ordering of the specific steps that need to be completed to produce the prototype system.

-- Determine benefit/cost/risk factors

If these factors are associated with each of the implementation steps, then the partial ordering information may be used to define a series of benchmark systems such that each partial system combines the highest benefits with the lowest cost/risk for that point in the overall development. The phases of the final implementation plan will be structured to produce the benchmark series of systems, with elapsed time and potential parallelism indicated for the phases.

-- Identify missing technology

Some of the identified implementation steps will undoubtedly depend on missing tools, techniques, or other knowledge. Along with the phased implementation plan, a parallel supportive research plan will also be developed.

AD-A126 101

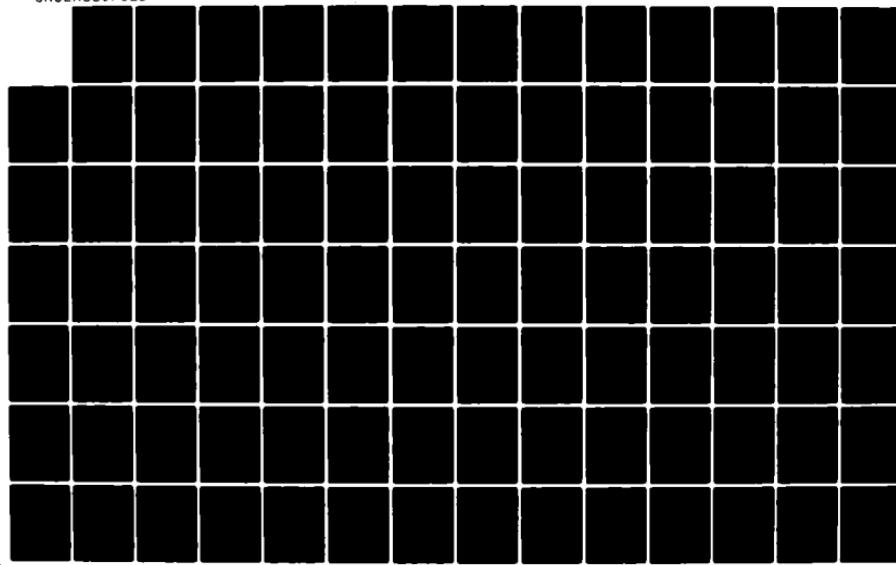
TOTAL SYSTEM DESIGN (TSD) METHODOLOGY ASSESSMENT (U)
WASHINGTON UNIV SEATTLE DEPT OF COMPUTER SCIENCE
G ROMAN ET AL. JAN 83 RADC-TR-82-331 F30602-80-C-0284

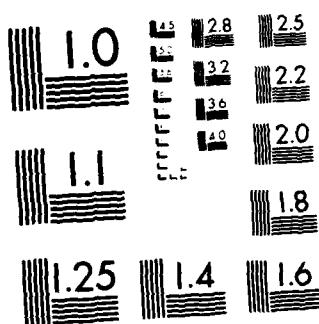
3/4

F/G 9/2

NL

UNCLASSIFIED





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963 A

OVERVIEW

This introduction has described the goal of cost-effective project support to be achieved by a TSD Facility. However, this goal may only be approached in stages, as described in terms of three initial objectives:

1. Characterize the TSD Environment.
2. Develop a design proposal for a TSD System.
3. Describe a phased implementation plan for a prototype system.

The balance of Section 4 describes the results obtained in trying to achieve these objectives. Section 4.2, Background, begins the task of characterizing the TSD Environment. Section 4.3, Proposal for a TSD System Family, completes the environment characterization and outlines the TSD System design proposal. Implementation plans are discussed in Section 4.4, Recommendations for Facility Development. A concluding discussion, Section 4.5, TSD Facility and System Design for DMA, presents the case for using the TSD System approach in the DMA work environment.

Note that the proposed TSD System Prototype design is based on a characterization of the TSD Environment, not on a set of detailed specifications. Thus the design is at a high level, allowing us to structure the "forest" before detailing the individual "trees". In other words, the system design proposal is for the overall system structure and organization. The implementation plan then describes the detail work that must be performed in order to bring the TSD System Prototype on-line. For example, the system requires the use of a command language and this fact is built into the design. However, the detailing of the command syntax and the specific command list is not included in this report.

4.2 BACKGROUND

There are many existing tools and environments. Characteristics of a few of the major systems are discussed in this section. However, an overall view may be obtained from work done at the Center for Programming Science and Technology, National Bureau of Standards. The Center has been compiling data on the availability of software development and testing tools [HOUNG80]. Their objectives are, first, to aid NBS efforts to develop guidelines and standards that will improve the quality of software. Secondly, to provide a means to determine what tools are available and what their capabilities are. Finally, to determine what tools are currently under development in order to share knowledge and avoid duplication of effort.

The collected tools are classified into one of five areas (data as of the October 1980 report):

1. Software Management, Control and Maintenance (110 entries).
2. Software Modeling and Simulation (7 entries).
3. Requirements/Design Specification, Analysis, and Program Generation (42 entries).
4. Source Program Testing and Analysis (96 entries).
5. Software Support System/Programming Environment (2 entries).

The extreme variation in the number of area entries suggests that either work has been concentrated on tools with the largest immediate benefits or that the easier problems were solved first.

PROGRAMMING ENVIRONMENTS

Unix [IVIE77, KERN81] (*) is a system that is widely used with great success. Many companies currently offer systems that are either derived from or compatible with the Unix system, and for processors ranging from microprocessors to large mainframes. A number of features help to explain its popularity, the most significant being the way files are handled, the way the command language is handled, and the way users of Unix (and its system language) have historically approached the problem of system development. The general style that has developed in the Unix community is unique and very productive for certain classes of users.

All files are treated uniformly by the Unix system. The files are assumed to be a sequence of bytes with no internal file structure, hence all structure is imposed by user programs and not by the operating system. Further, the file system even treats all peripheral devices as files. From the programmer's standpoint, the homogeneity of files and peripheral devices is a great simplification.

When a user logs into a Unix system, a command interpreter called the "shell" accepts commands and interprets them as requests to run programs. The user's terminal is just another file in the file system. Part of the

(*) Unix is a trademark of Bell Laboratories.

function of the shell is to connect the input/output files referenced by a program to the actual files supplied. This allows any program to use input from the user's terminal, an actual file, or the output from another program. The shell capability, plus the ability to easily pass information between programs (no special arrangement is necessary!), has led Unix users to develop a style in which each program is specialized to just one task. Programs and programming thus tend to be much simpler than if each program attempted more.

In the Unix environment the average program is rather small. People tend to search for ways to use existing tools instead of laboriously making new ones from scratch. Thus we have a model environment in which large systems may be constructed easily from many small pieces by supplying the appropriate interconnections. In addition, system modifications may be made by replacing individual specialized pieces without upsetting the overall program operation.

Interlisp [TEIT81] is a programming environment, based on Lisp, that is in widespread use in the artificial intelligence community. The nature of this user group has greatly influenced the characteristics of the system. The typical users are engaged in experimental rather than production programming. They were willing to expend computer resources to improve human productivity, and they prefered sophisticated tools even at the expense of simplicity.

A program may undergo drastic revisions, in experimental work, as the problem being solved becomes better understood and more completely defined. Keeping track of such change for a large program is an extremely complex task. It is the job of the Interlisp file package to help the programmer manage this task by automating the necessary bookkeeping of where things are and what things have changed. The file package supports the abstraction that the user is truly manipulating his program as data and that the file is merely one particular representation of a collection of program pieces.

An impressive feature of Interlisp is the "DWIM" (Do What I Mean) facility. It is invoked when the basic system detects an error, and it then attempts to guess what the user might have intended. Thus spelling error corrections, command corrections, misspelled function name corrections, and other such corrections can all be achieved within the Interlisp environment. This type of support helps to provide the desired increase in user productivity, but at the expense of a significant amount of computer resources.

Interlisp has been characterized as friendly, cooperative, and forgiving, at least for the skilled users. However, the two attributes that set it apart are the degree to which the system is integrated and the degree to which the facilities can be tailored or extended. The integration means that any facility may be called from any other facility. For example, the editor may be called from inside the debugger. The various facilities readily call on each other for important support during a session, which means that the integration of the facilities actually increases their power. Interlisp provides for extensions by allowing the user to specify a function to be called whenever a facility encounters any

object, expression, or command that it does not recognize. Alternately, almost all facilities support extensions via substitution macros which associate a template of existing commands with a new command.

Interlisp has an abundance of user setable parameters which allow it to support a wide variety of programming styles. However, it has been carried to the point that new users are usually overwhelmed and intimidated by the sheer number of choices that must be made.

The Ada environment (Stoneman) [STEN81, WOLF81], in contrast to Unix and Interlisp, is not a production system as yet. However, a considerable amount of work has already gone into its specification. A primary requirement is to allow a project team always to work in terms of the Ada language, rather than in terms of a particular target machine. Thus the environment should offer comprehensive support for the full Ada language.

Another major objective of this environment is to offer effective support to a project throughout its life cycle, from initial requirements specification through long-term maintenance. This means that the project database must be able to hold all relevant project information (source code, binary code, documentation, test histories, etc.), as well as maintaining accurate records of relationships among the items of information as the project evolves. It also requires that the environment must provide for configuration control and management control.

Stoneman represents a commitment to an open-ended environment. That is, the tool set included in the system may be modified or extended at any time. The individual can develop tools that support his own style of working. Of course, there is a danger in that this may lead to lack of portability, but it also raises a more serious question. Can there be a complete and accurate database recording of the relationships between objects when these objects are created by user-supplied tools?

For reasons of portability, Stoneman recognizes three distinct levels within the environment. These consist of the kernel Ada programming support environment (KAPSE), the minimal Ada programming support environment (MAPSE), and the full Ada programming support environment (APSE). The KAPSE is a system and tool portability level, and the MAPSE is a user portability level. An APSE is based on a particular MAPSE, but may include additional tools that support use of specific methodologies.

Ada is a machine independent language, but that is not sufficient for tool portability. The language definition does not address such issues as the organization of the database or the means for tool composition. Portability requires the definition of some framework in which tool programs execute, and the KAPSE provides this framework.

The MAPSE consists of a minimal comprehensive tool set, in that no smaller tool set is adequate for the purpose and that it is possible for all members of the project team to work with just this tool set at all stages of the life cycle. An APSE can incorporate additional tools of general interest, of interest to a particular project only, or of interest to just one individual programmer. In the degenerate case a MAPSE is itself a APSE. While a MAPSE offers very general tools, an APSE can be

more specific and contain tools that encourage, or even enforce, use of a particular design or development methodology.

DESIGN ENVIRONMENTS

PSL/PSA [TEIC77] is a system concerned with one approach to improving systems development. The approach is based on three premises: first, that more effort should be devoted to the front end of the process where a proposed system is being described from the user's point of view; second, that the computer should be used in the development process since systems development involves large amounts of information processing; third, that a computer aided approach to systems development must start with documentation.

In computer aided logical system design, the object is to produce the System Definition Report. The capability to describe systems in a computer processible form results from using the Problem Statement Language (PSL). The ability to record such descriptions in a database, incrementally modify it, and on demand perform analysis and produce reports, comes from the software package called the Problem Statement Analyzer (PSA). The use of PSL/PSA does not depend on any particular structure of the systems development process or any standards on the format and content of hard copy documentation.

PSL is based on a relatively simple model of a general system. It states that a system consists of things which are called OBJECTS. These objects may have PROPERTIES, and each of these properties may have PROPERTY VALUES. The objects may be connected or interrelated in various ways by connections called RELATIONSHIPS. The general model is specialized for an information system by allowing the use of only a limited number of predefined objects, properties, and relationships.

The system design activities assumed and supported by the PSL/PSA design environment include:

1. Data Collection: since most of the data must be obtained initially through personal contact, interviews will still be required. The data collected are recorded using PSL.
2. Analysis: a number of different kinds of analysis can be performed on demand by PSA and therefore need no longer be done manually.
3. Design: design is essentially a creative process and cannot be automated. However, PSA can make data available to the designer and allow him to manipulate it extensively. The results of his decisions are also entered into the database.
4. Evaluation: PSA provides some rudimentary facilities for computing work measures from the data in the problem statement.

5. Improvements: identification of areas for possible improvements is also a creative task. However, PSA output, particularly from the evaluation phase, should be useful to the analyst.

Thus the System Definition Report ultimately contains a large amount of material produced automatically from the design database.

The Software Requirements Engineering Methodology (SREM) program [BELL77], is a system that includes techniques and procedures for requirements decomposition and for managing the requirements development process. A major part of SREM is the Requirements Engineering and Validation System (REVS), a computer-aided system to support the requirements development activities. REVS consists of three major segments: a translator for the Requirements Statement Language (RSL), a centralized data base called the Abstract System Semantic Model (ASSM), and a set of automated tools for processing the information in the ASSM.

RSL is designed to be a means for stating requirements naturally while still being rigorous enough for machine interpretation. It is oriented around the specification of flow graphs of the required processing steps expressed in terms of four primitives: Elements, Relationships, Attributes, and Structures. The language is extensible at the concept level by adding new types of elements, relationships, and attributes. This extensibility allows the language to respond to application specific needs and other unanticipated needs for stating requirements. The RSL statements are input to REVS through a translator that checks the statements for individual correctness, and then abstracts them. The extracted information is then entered into the ASSM. No executable code is generated, only the entries in the data base that will later be used by other REVS tools.

The information available in the ASSM will support a wide variety of analysis tools. Normally available is a baseline set of widely applicable tools which perform analyses primarily related to flow properties of the information in the problem specifications. This capability is very important for generating consistent, correct requirements and enforcing any desired discipline on the requirements generation process. Among the available tools are an interactive graphics package to aid in the specification of the flow paths, static consistency checkers which check for consistency in the use of information throughout the system, and an automated simulator generator and execution package which aids in the study of dynamic interactions of the various requirements.

FACILITIES

The facility concept has been implemented or proposed in a number of cases. Three of these are considered as a context for the proposed TSD System.

The System Architecture Evaluation Facility (SAEF) [ANDE78,CLARK], is located at RADC. SAEF is designed to provide an experimental laboratory for research into the advanced hardware configuration necessary to support the complex data processing needs of military command, control, and

communication systems. It allows overall system designs and alternatives, both hardware and software, to be quickly and easily evaluated, thus minimizing actual development and life-cycle costs for new systems.

Direct experimentation with unique hardware architectures is extremely expensive and time consuming. It is also wasteful of resources because prototypes are rarely useable systems and hence are discarded after the initial studies are completed. Rather than actually build new hardware components, SAEF provides the means by which they may be emulated by a microprogrammable computer. In effect, the microprogrammable computer (a Nanodata QM-1) is molded to look and act like the proposed design at the instruction set level. Thus, machine language level programs may be written for the proposed machine design and then executed by the microprogrammed machine emulation.

The QM-1 is operable in both a stand alone mode (where it supports a full complement of peripherals) and in a time share mode connected to a DECsystem-20 computer. Since many support tools are essential to the successful operation of such a facility, many of the tools run on the separate DECsystem-20. These tools provide the necessary control and reporting capabilities for the installation complex, plus allowing a convenient user interface to the facility.

There are two major problems in the SAEF approach that require specialized tools: one is to define the hardware system architecture to be emulated, and the other is to generate code to run on the defined machine. SMITE (Software Machine Implementation Tool for Emulation) was developed to attack the first problem. It is a high order language which allows machine descriptions at the register transfer level. These descriptions are compiled into microcode to run on the QM-1. Once the description of an architecture has been implemented through SMITE and its associated support software, there is a need to write software for the emulated machine. What is needed is a compiler that would automatically compile object code for a machine based on the SMITE description of that machine. Such a retargetable compiler is still subject to research and development efforts.

CAMEO [RYAN79] is a system being developed to provide a single, integrated capability for a dual QM-1 computer configuration in which the design and performance characteristics of definable target systems can be easily represented. This representation is in terms of a model consisting of one or more interacting emulations and/or simulations. CAMEO stands for "Concurrent Application of Multiple Emulations On-line", and it provides for all user access to the QM-1 through a common standardized environment.

Based partly on the importance of configuration management practices and partly on the need for organizational and accounting controls, all users interact with the CAMEO system through "Target Systems Complexes". These complexes are created and maintained as an integral part of the CAMEO data base. One or more target systems may be prepared for exclusive access under each complex.

In addition to the facility to control target systems, software may be developed at any of three levels:

1. Programs prepared, debugged, and applied in the context of a target system. This is a function of the facilities of a pre-existing target system with an operating system and its utilities.
2. Programs prepared, debugged, and installed to function as an operating system for a target machine being emulated. This is supported by appropriate CAMEO utilities.
3. Programs prepared, debugged, and applied in the CAMEO System context to function as a new target machine emulation or simulation.

The CAMEO System development goals call for capabilities which are in some respects conflicting: there is a stated critical need for an operating environment in which the user's target system can perform realistically at near real-time speeds; on the other hand, facilities must be provided through which the users can deal effectively with the problems of software generation, testing, and performance evaluation.

FASP, the Facility for Automated Software Production, is a Naval Air Development Center facility in which operational and system test software for any Navy platform can be developed and maintained. This facility offers a large complex of commercial computers, including two CDC 6600's and one CDC CYBER 175, plus extensive supporting peripheral devices. The software supported includes a full suite of program-generation capabilities for standard Navy languages and target machines, together with tools to support the use of modern software engineering technology. As a software generation facility, FASP complements the capabilities of individual platform integration facilities where the operational hardware configuration is mocked-up in a simulated environment for testing, evaluation and training.

The goal of FASP is to keep pace with emerging software engineering methodologies and tools, and to provide support for the Navy's standard military computers and standard high order languages, assembly languages, and microprogram languages.

FASP is a comprehensive software generation facility consisting of an integrated collection of software development and maintenance tools. The tools are designed to provide support for each phase of the software life cycle: (1) design, requirements and specification aids, (2) implementation tools (translators and system generators), (3) testing tools, (4) project management tools, and (5) configuration management tools.

The remote terminal capability of FASP allows Industry, Navy Laboratories, and other Government Agencies to use FASP for software production and maintenance. FASP then insures continuity from system development by the prime contractor through maintenance by the Navy. The

same tools and records that support the development remain available throughout the maintenance phase, thereby minimizing transition problems and training.

COMPARISON WITH TSD OBJECTIVES

One useful dimension for comparing facilities comes from the evaluation of the environments supported by the facilities. Cheatham [CHEA80] suggests a comparison based on the following functional aspects of a facility:

- Language Support
- Target Configuration Support
- User Interface
- Command Language
- Integration of Tools
- Granularity of Tools
- Relationships Supported
- Protection
- Documentation Support
- Management Support

All of the systems discussed in this section, both existing and proposed, adequately satisfy the TSD System objectives for some of these areas. However, none of the systems adequately support the entire range. A top-level characterization of the TSD System is presented in the next section, where these functional aspects are treated in more detail.

REFERENCES

[ANDE78] Anderson, D. R., Anderson, L. D., and Wen, K. Y., "Reconfigurable Computer System Design Facility," Report RADC-TR-78-6, Rome Air Development Center, Griffiss Air Force Base, NY 13441, January 1978. Vol I, AD#A052 995; Vol II, AD#A053 013

[BELL77] Bell, T. E., Bixler, D. C., and Dyer, M. E., "An Extendable Approach to Computer-Aided Software Requirements Engineering," IEEE Trans. Software Engineering, SE-3, No. 1, pp. 49-60, January 1977.

[CHEA80] Cheatham, T. E., Jr., "Comparing Programming Support Environments," Software Engineering Environments, Hunke, H. (editor), North-Holland, 1980, pp. 11-25.

[CLARK] Clark, Capt N. B. and Troutman, 2Lt M. A., "The System Architecture Evaluation Facility, an Emulation Facility at Rome Air Development Center," White Paper, RADC.

[IVIE77] Ivie, E. L., "The Programmer's Workbench - A Machine for Software Development," Comm ACM, 20, No. 10, pp. 746-753, October 1977.

[KERN81] Kernighan, B. W. and Mashey, J. R., "The UNIX Programming Environment," Computer, 14, No. 4, pp. 12-24, April 1981.

[RYAN79] Ryan, R. H., CAMEO: Concurrent Application of Multiple Emulation On-Line, Report MDC G7958, McDonnell Douglas Astronautics Company, Huntington Beach, Calif. 92647, February 1979.

[STEN81] Stenning, V., et al, "The Ada Environment: A Perspective," Computer, 14, No. 6, pp. 26-36, June 1981.

[TEIC77] Teichroew, D. and Hershey, E. A., "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems," IEEE Trans Software Engineering, SE-3, No. 1, January 1977.

[TEIT81] Teitelman, W. and Masinter, L., "The Interlisp Programming Environment," Computer, 14, No. 4, pp. 25-33, April 1981.

[WOLF81] Wolfe, M. I., et al, "The Ada Language," Computer, 14, No. 6, pp. 37-45, June 1981.

4.3 PROPOSAL FOR A TSD SYSTEM FAMILY

INTRODUCTION

In this section we shall determine significant functional properties of a TSD System by characterizing the nature of the desired TSD Environment. This effort will result in a top level view of the TSD System requirements and a corresponding set of system specifications. The section concludes with a TSD System design proposal for a prototype system that will meet all of the stated requirements at the top level.

In the discussion to follow we shall adopt the view that only one project *must* be supported. This approach is based on the assumption that two different projects are independent, and hence will be developed in completely separate environments. The possibility of joining separate projects (or their supporting environments) is viewed as a higher-level management/facility function that is not to be considered here. Of course, various parts of a single project may be in different life cycle stages at the same time, and hence these stages must all be supported simultaneously.

CHARACTERIZATION OF THE TSD ENVIRONMENT

The Scope of the System

A complete environment for a specific application provides tools to support the complete application system life cycle. One element in the evaluation of any proposed system is to establish how well each area is supported by tools and how thoroughly the support is integrated across the areas. The following suggested list of functional areas is given in [HUNK80]:

- requirements analysis and specification
- system specification
- project management
- implementation, coding, and testing
- simulation and modeling
- system integration
- cost estimation and cost control
- verification, validation, and inspection
- configuration control and version control
- acceptance testing
- modification and maintenance

As discussed in Section 4.1, the goal for the initial TSD Prototype System is to provide an environment that supports only the first three stages of the project life cycle, thus the final five functional areas in Hunke's list will not be considered at this time. However, this proposal is written with the understanding that extending the TSD System to cover the entire life cycle may be eventually economically justifiable, and hence must be allowed for with a minimum of system re-design.

It is the purpose of the TSD Environment to support and encourage the use of TSD Methodologies by all users of the system. This implies that the level of integration of the system parts will be sufficient to discourage random or disorganized use of its capabilities, while strongly encouraging organized and purposeful use. To this end, and following the TSD Methodology concepts, the system must be easily adapted to the specialized needs of any specific application or application dependent methodology. Organizational concepts from the TSD Framework must be built into the structure of the environment, but otherwise it must be extremely flexible in style and detail.

The Human Interface

[SPIE80] suggests that a friendly user interface for an environment must allow the user to create and use new private tools specifically geared to the user's needs of the moment. A variation of this requirement is to allow the user to superimpose a private interface on an existing standard tool. All of this so that the user is not forced into the work-patterns envisioned by the environment/tool designer. [PREN81] suggests that the environment should be adaptable, user-centered, suggestive, helpful and supportive, and not imposing. User friendliness should also include human interfaces other than text, such as menu selection capability, graphics, and possibly voice recognition.

There generally will be three classes of users of the TSD System: managers, designers, and evaluators. Each user class may be further divided into the naive and the sophisticated users. Each set of users will need their own special tools and support utilities, as well as access to general tools needed by all system users. By providing a tailored environment for each of these groups it will be possible to provide the type of friendly interface to the system that actively encourages the use of the system. Thus three standard environments should be provided by default: a management environment, a design environment, and a production environment. In addition, the ability to define and add additional environments must be readily available to the sophisticated user.

The management environment provides the appropriate tools to support all project management tasks, including configuration and version control. The design environment supports the project requirements, specifications, design, and programming activities. The production environment is the most non-traditional in orientation: it is responsible for the testing, simulation, and modeling activities that lead eventually to system integration and final acceptance testing.

Types of Tools

The tight coupling between a project, its appropriate methodology and its support environment, and the users of that environment suggests that each project needs a different environment. In particular, it is important that the tools provided by the environment be continuously supportive to the users in their day to day work. Perhaps the environment should even change continuously as a project evolves through its life-cycle, but too much or too rapid a change in a system would present problems in human learning and adaptation. Even so, could we possibly

build every possible environment from scratch as we need it? Yes, but the effort required to do so would be tremendous, since we would be, in effect, re-inventing many parts of the wheel for each new environment. Therefore, to minimize the effort required to create an environment, it is necessary to configure environments largely from standard modular capabilities. The problem is to balance the advantages of using standardized broad-ranging tools for all applications against the advantages of specializing each tool to a particular application.

Some types of tools will be needed for all applications. For example, information about the developing system must be stored in a project database so that the appropriate tools, under the command of the users, may study and transform the project information. Thus a central TSD System database must be defined and the necessary tool interfaces must be created. It is assumed that all project information must be stored in the automated project database in order to make the TSD System truly supportive of all of the users. Such a total depository of information requires an equally total control of access to insure the integrity and security of the data.

Other types of tools may be more specialized for particular environments. Text processors, for example, should be tailored to the users and the application: helpful for beginners, terse and powerful for experienced users, able to process straight English text for reports and documentation, able to process partially formal text for problem requirements definition, and able to process formal text for a rigidly structured programming language. Implementation of such diversity could come from one generalized tool driven by appropriate data tables ([STAL81] describes an elegant use of this approach to develop an editor with many of these properties).

Tools directly affected by the definition of the target system represent a somewhat different case. For example, language compilers have a front-end that depends only on the source language being used (FORTRAN, Ada, etc.), and a back-end that depends only on the target machine architecture (DECsystem-20, VYK-32, etc.). If all languages and all machines were known and fixed, then a complete set of compilers could be written and incorporated into the support environments. However, this is simply not the case, even though it has been the traditional approach. By defining the correct level of detail for tool modules, pieces may be assembled to produce the necessary final results. Thus a FORTRAN compiler front-end could be assembled with a back-end code generator for a new machine, producing a new FORTRAN compiler. The production of the new compiler requires defining new code generators with a standard interface, not the creation of an entirely new program.

The total tool set available in the TSD System prototype must support the functional areas of: project management; requirements analysis and specification; systems specification (both hardware and software); implementation, coding, and testing; system integration; simulation, emulation, and modeling. These functional areas, and the corresponding tools, are distributed to the appropriate environments in order to provide the tool users with a system context. It is the TSD System itself, through control of the environments, that defines the system context that

ensures each user is provided with all of the tools and data necessary (and no more!) for the user to function in a productive cost-effective manner.

Tool Integration

In an important sense, the basic function of an environment is to ensure that everybody connected with a project gets the information that they need, when they need it, and that the results of their work are preserved. This implies that at the center of the environment there must be a database containing all of the information about the project. The existence of the project database does not guarantee the integration of the various tools in the TSD System, but it does make tool integration possible.

Given a complex system of cooperating tools that use a supporting database, where and how does the user interact? There must be a command language so that the user may give instructions to the system. The command language should be as uniform and general as possible to satisfy portability considerations. The TSD System, through the command language, should look the same, as much as possible, to all users located at all installations. The command language should contain no surprises; that is, any command should do what you would normally expect such a command to do. Further, all commands should be failsafe in the sense that it should not be possible for a slight mistake to have catastrophic consequences (you said "delete", and an entire file was deleted instead of just the last line!). The commands should also be self-documenting. The naive user should be able to ask at any time "What does this mean?", or "What options are available to me now?", and yet the sophisticated user should not be impeded by such a facility.

It is important for a user to not be distracted or surprised by the system reacting to features or tools that the user does not know about. Thus a significant feature of an environment is the tool access rights granted to a user of that environment. A manager using a management environment should not have to be concerned with tool or system names used by a designer operating in a designer environment. In general, the defining of environments and the corresponding granting of tool access rights is a project management function.

In a similar and perhaps even more important manner access rights to the project database must also be controlled. Operational questions must be answered, such as: How does a designer make a temporary change to a file in order to test the change, and yet not affect any other person that may be using the same file? When and how is a temporary change in the system incorporated as a permanent change? How is the version/level problem to be solved for the given application? The database access control must provide the mechanism to solve these problems. Further, it must also control tool access to the database, since a user command may trigger a series of unforeseen tool operations (read, write, update, etc.) that may not be desired.

The integration of tools essentially allows any tool to work with any other tool, or to manipulate the database. The power and scope of this concept is controlled in two stages: (1) limit user access to tools, and (2) limit user and tool access to the database.

Dynamics of the TSD Environment

Certain tools are of such common utility that all environments are expected to use them, independent of whatever application area is being considered. These tools are elements of the core tool set. Text processors and database utilities, for example, will always be needed and hence are in the core. The ability to define and create environments, and to authorize the use of those environments and any associated parts of the database must also be functions performed by tools in the core. The core will always represent the common tool set for all TSD Systems.

Anyone that has access to the core tools that define a new environment may use those tools. That is, they may actually define a new sub-environment of whatever environments they originally could access. This automatically allows a network accessing structure, since a project manager can (by definition?) always access any information about the entire project, and the manager may create and allocate any number of sub-environments for each group leader in the organization. The group leaders may, in turn, create and allocate further sub-environments for each of their group members. Thus the necessary overlapping and restricted spheres of access may be established as appropriate to the given application and project management organization.

A TSD SYSTEM DESIGN PROPOSAL

Figure 4-1 is a top-level diagram of the proposed TSD System design which incorporates all of the previously discussed factors. The main control module of the system is the Command Language Interpreter (CLI). All user communication must pass through this module for interpretation, control, and possible execution. The CLI keeps track of all system resources and user authorizations, allowing it to automatically provide users with their proper environment when they log onto the system.

User communication with the TSD System is through work stations (WS₁,...,WS_n). These stations may be as simple as dumb CRT terminals, or as complex as sophisticated graphics work stations, depending on the user/application requirements.

Each TSD System may also be connected to selected specialized peripherals (SP₁,...,SP_m). Although every installation will have standard peripherals, such as line printers and tape drives, more specialized devices such as those needed for emulation (perhaps a QM-1) or VLSI design (perhaps a large plotter) may not be available so universally. In order to share the use of such devices, one of the specialized peripherals is assumed to be a network connection. This allows a user from one TSD System to access and use the specialized devices that may be available at another installation. As indicated in Figure 4-1, all such accessing goes through the appropriate interface routines (T₁,...,T_m), but the central

control still resides in the respective CLI modules.

All communications with the project database must pass through the Database Access Control (DAC) module. The DAC uses the appropriate project database tables to either allow or disallow the requested user/tool access. In order to provide for portability these access requests assume a standardized TSD System database structure. However, this is essentially a "pseudo" or "logical" database in that it probably will be more economical to use an existing commercial database system for the actual physical data storage and retrieval operations. Thus the TSD database management system may be viewed as an interface between the assumed TSD logical database and the actual physical database. Note that this means that different commercial database management systems could be used at different TSD System installations, with the only added expense that of redefining the TSD DBMS interface.

The CLI uses the Tool Access Control (TAC) module to maintain control over the use of the tool set. User requests for tools, tool requests for tools, tool compatibility and interface requirements, and all other information about tool usage will be maintained through this module. The TAC essentially maintains the integrity of the system integration. New tools to be added to the system are integrated into the system by supplying the appropriate information to the TAC module.

The Core Tools module represents the collection of all standard TSD System tools available at any given time. As mentioned previously, some installations may have additional specialized peripherals that require unique tools. In general, the tools that are unique to a given TSD System installation are contained in the Other Tools module.

CONCLUSIONS

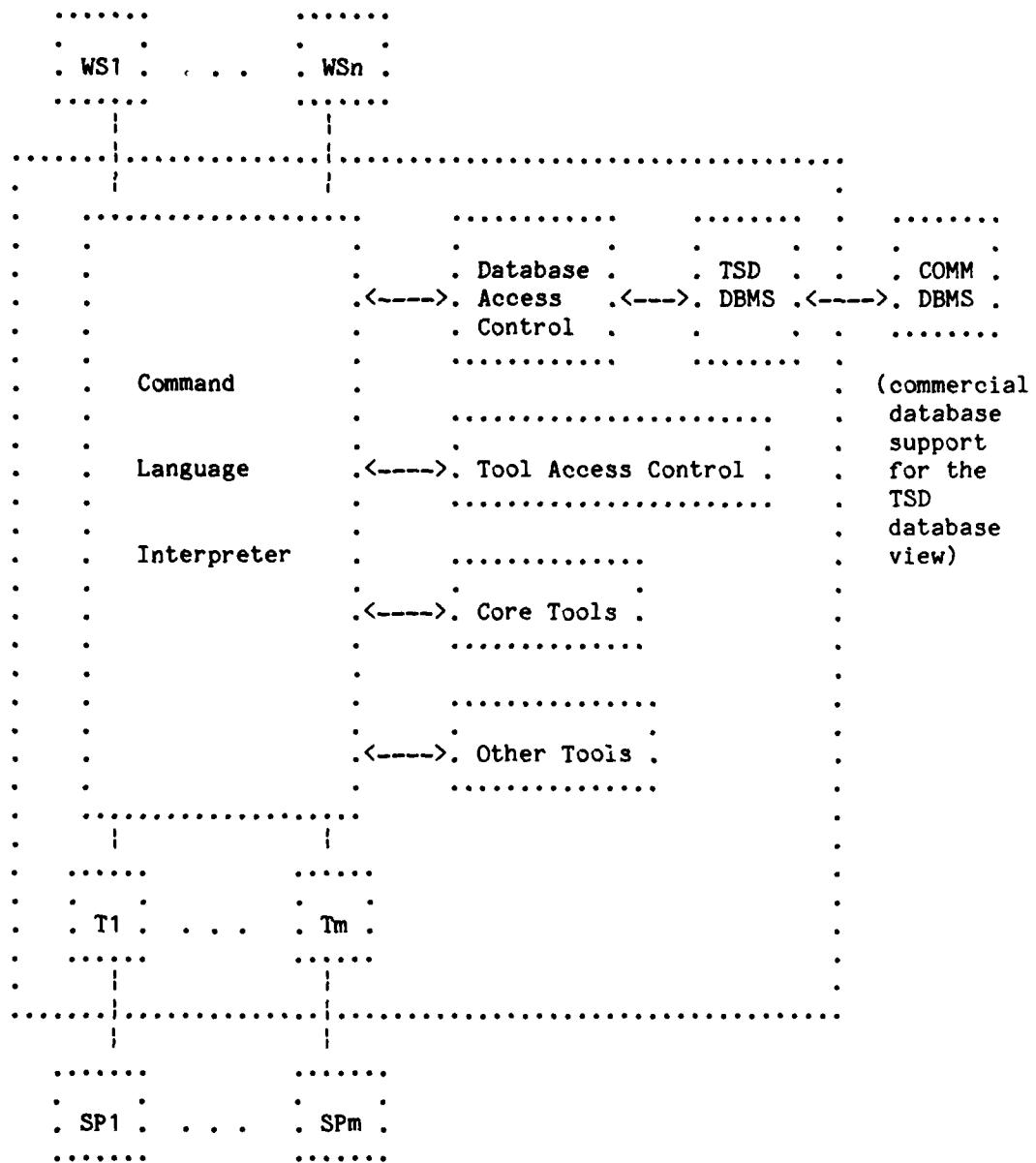
One poorly understood aspect of many current environments and facilities is what has been termed the "gulp factor" [KERN81]. This concept deals with what must be done to adopt a new environment. Many current environments are high on the gulp factor scale because they must be adopted all at once, require a massive retraining of all potential users, and/or provide little support for systems previously developed. Although the TSD System design proposal has been presented as a single stand-alone entity, the background considerations that went into its development were based partly on making the maximum use of existing resources and partly on making the transition to a TSD Environment as easy and desirable as possible. That is, the envisioned eventual implementation was designed to be low on the gulp factor scale.

The high level design proposal presented in this section represents the ultimate goal to be achieved by the implementation plan described in the next section.

REFERENCES

- [HUNK80] Hunke, H., "Introduction," Software Engineering Environments, Hunke, H. (editor), North-Holland, 1980, pp. 1-9.
- [KERN81] Kernighan, B. W. and Mashey, J. R., "The UNIX Programming Environment," Computer, 14, No. 4, pp. 12-24, April 1981.
- [PREN81] Prentice, D., "An Analysis of Software Development Environments," ACM SIGSOFT Software Engineering Notes, 6, No. 5, pp. 19-27, October 1981.
- [SPIE80] Spier, M. J., Gutz, S., and Wasserman, A. I., "The Ergonomics of Software Engineering - Description of the Problem Space," Software Engineering Environments, Hunke, H. (editor), North-Holland, 1980, pp. 223-234.
- [STAL81] Stallman, R. M., "EMACS, the Extensible, Customizable Self-Documenting Display Editor," MIT Artificial Intelligence Laboratory, AI Memo 519a, March 1981.

(Work Stations Offering Specialized
Design and Management Environments)



(Specialized Peripherals, such as emulation machines)

Figure 4-1. TSD SYSTEM HIGH LEVEL DESIGN

4.4 RECOMMENDATIONS FOR FACILITY DEVELOPMENT

INTRODUCTION

One of the main goals of the TSD Facility development effort is to enhance the productivity and the quality of system design efforts related to DMA production. This goal is to be achieved through supporting the use of systematic design approaches in a cost-effective manner. In establishing a plan to implement this support it was found that the resulting cost-effectiveness could be enhanced considerably by creating both a short term and a long term development plan. These plans, in combination, form the TSD Facility Development Master Plan.

The short term view of the TSD Facility represents a low-risk, high-benefit utilization of mostly available resources. It is organized in such a way as to provide immediate production support. In addition, it serves as a compatible "hot-bench" for the long term research and development necessary for the Facility to achieve its ultimate goals.

NEAR TERM FACILITY DESCRIPTION

There are a number of distinct components that make up a prototype TSD Facility:

- TSD System
- Physical Resources
- Technical Staff
- Management Staff

Each of these components must be defined and in place for the operation of a successful facility. The following discussion points up some of the aspects of these components that need to be established in the near term. The purpose of this discussion is to help describe the expected nature of the near term prototype TSD Facility as produced by the Facility Development Master Plan (Figure 4-2).

The TSD System is composed of software, with perhaps the addition of some specialized hardware. It is assumed that the TSD System executes under a standard operating system. The operating system is assumed to be available on whatever computer complex is used for the prototype TSD Facility, and that it is complete with a normal complement of utility routines. Terminal and network communication protocols are also assumed to be available.

A detailed set of requirements and a corresponding set of specifications must be established for the TSD Environment. (The study must consider the relation between the Ada Environment specifications and the TSD System effort.) The initial design of the command language and the design and implementation of the prototype command language interpreter must be accomplished. The unified logical database must be defined, along with the data accessing mechanisms to be used by the command language interpreter. Techniques for interfacing existing and

proposed tools with each other and with the project database must be defined.

The TSD System must provide support for TSD Methodologies for some number of applications. This means that specialized languages for system requirements and system design must be developed for these applications, as well as appropriate analysis techniques. The TSD System must then integrate and support the new tools that make these languages and techniques available to the users.

Currently available support for specialized tasks must also be extended and integrated into the TSD System. For example, the analysis of most embedded systems requires the use of high speed simulation or emulation studies. The tools that support these studies need to be augmented in order to be more responsive to the increased levels of distribution and complexity encountered in current advanced architectures. (SMITE, for instance, needs to be enhanced to provide for distributed hardware and multilevel designs.) The future use of Ada in DoD projects must be identified, and its role in the evolution of the TSD System must also be defined.

Physical Resources

In the near term there is not sufficient time for a major equipment procurement, so it is expected that existing computer installations will have to serve as hosts for the prototype TSD Facility. Questions must be resolved concerning the sharing of physical resources for the TSD Facility versus resources dedicated to the Facility.

Technical Staff

The development of the TSD Facility represents a major technical effort that must be supported by an appropriate organization of technically trained personnel.

First, a significant factor in the success of such an undertaking is the user assistance provided by the staff. No matter how friendly and helpful the on-line environment may be, both prospective and active users need well written introductory and advance guides for using the system. Experts are needed that may teach application oriented courses in the use of the system, or simply be available to answer questions from puzzled users. Information about the Facility availability must be widely distributed to potential customers.

Another significant technical activity is in the area of development, maintenance, and enhancement of the existing TSD Facility. Much of this effort will be based on feed-back from active users, to correct problems encountered or add features not already implemented. Porting of the TSD System to alternate facilities must also be considered.

Research into TSD System related topics should also be an on-going effort in order to maintain the evolution of the system to its state-of-the-art, most cost-effective form. Perhaps some of this effort should be guided by feed-back from prospective users that refuse to become

active users.

Management Staff

Coordinating, planning, and monitoring of all of the tasks described in this section requires a management effort with an appropriate support organization.

FIVE YEAR PLAN

Figure 4-2 details the steps in the TSD Facility Master Plan. The following explicit objectives have determined the nature of that plan:

- low development cost;
- speedy development;
- limited risk;
- early availability to potential users;
- ability to respond to immediate design needs without compromising the long range requirements;
- smooth growth in capability and range of applicability;
- compatibility with other related DoD efforts (e.g., SAEF, Ada);
- strong interaction between R&D and production efforts.

Because the development of a design facility is generally a high risk and high cost proposition, the strategy adopted in the master plan is to minimize new tool development and focus on integrating off-the-shelf components to the greatest possible extent. While the command language, database view and core tools which characterize the central part of the TSD Environment could be assembled together from existing components, the lack of application specific tools could make it difficult to attract potential users of the prototype TSD Facility. This may be avoided if an already successful existing facility could be used to supply the application oriented tools. SAEF has been selected to meet this objective. This particular choice has several other advantages. It employs a facility which is compatible with the general TSD Concept, it provides continuity to the entire TSD program, it addresses a class of users who feel most acutely the need for a design facility (for embedded systems), and it promises immediate and high payoffs.

The result of these and other considerations is a plan which consists of three concurrent efforts which gradually merge into one. The main stream deals with the selection and integration of the TSD Facility components. The other two focus, respectively, on increasing the effectiveness of the application specific tools through enhancements to the SAEF and on providing the technical support needed for long range planning through the development and evaluation of new TSD Methodologies.

The development and evaluation of new TSD Methodologies is meant to have little or no impact on the near term version of the TSD Facility. The objective is to assist in the later evaluation and subsequent enhancements of the TSD Facility available at the end of this planning period. This is to be accomplished by developing tools to be incorporated in subsequent versions of the facility and methodologies that define the

manner in which such tools should be used in various application areas. Because effective methodologies are application dependent, the plan suggests work to be concentrated only on a few application areas of special significance within DoD. Corresponding independent refinements of the TSD Methodologies should be produced for each selected area. Following the methodology development, empirical evaluations on real-life moderately sized projects should be carried out. The experience should be used to further refine and tune the methodologies to the needs of the respective application areas. The development of specification languages and analysis techniques should be centered around mechanizing some of the activities involved in applying the methodologies. This is the point where some integration between the intentionally independent undertakings ought to take place. The level of effort required by this particular stream of the master plan depends upon the range of applications being chosen. (No more than three areas should be attempted.)

SAEF enhancements are motivated by the desire to make the ultimate facility more attractive to potential users, to build a user community concurrently with the development of the facility, and to establish a realistic base for determining the priority assigned to introducing various core tools. Since it is expected that not all core tools will be available in the prototype facility, those tools that appear to be most needed by the particular community of users ought to be considered first. Furthermore, current understanding of the specification language needs for the system design stage should be used in the design of the next version of the hardware description language used by SAEF. This stream of activities is also independent in nature from the other two.

The main thread of the master plan is concerned with building a TSD System from available components and its integration with the SAEF to form the TSD Facility prototype. The approach is actually consistent with the TSD Methodologies. It starts with the problem definition stage during which a detailed definition of the TSD Environment (only outlined by this study) is generated. Based on the TSD Environment definition a system architecture for the TSD System is developed in a manner which is consistent with the constraint that the proposed architecture must be supported primarily by the resources available in SAEF. (Given the short range nature of the plan hardware procurement ought to be avoided.) Next the binding phase is carried out. It consists of the selection of existing tools required to support various entities of the TSD System and of the definition of custom software needed to integrate them. This activity represents, in the terminology of the TSD Framework, the generation of software requirements. (The hardware is given in this case.) The integration of the tools is carried out in stages. The last one involves placing all acquired tools on the SAEF and thus establishing the TSD Facility prototype. Once some experience with the use of the TSD Facility on several production efforts is accumulated, it is time to reevaluate the facility and to devise new plans for its future.

CONCLUSIONS

Three concurrent time lines of activities have been defined in order to maximize both the short term utility and the long term benefits of the prototype TSD Facility development effort. The final merging of the separate time lines produces a prototype TSD Facility that has demonstrated its ability to achieve the original goals in routine production use.

REFINEMENT OF TSD METHODOLOGIES FOR SEVERAL KEY APPLICATIONS	ENHANCEMENTS TO SAEF CAPABILITIES	DEVELOPMENT OF TSD SYSTEM PROTOTYPE
SELECTION OF KEY APPLICATIONS	STUDY OF POTENTIAL SAEF ENHANCEMENTS	DETAILED DEFINITION OF TSD ENVIRONMENT AND SYSTEM
DEVELOPMENT OF NARROWLY FOCUSED TSD METHODOLOGIES	UPGRADING OF SAEF CAPABILITIES	TSD SYSTEM BINDING (TOOL SELECTION)
EMPIRICAL EVALUATION OF TSD METHODOLOGIES	INTEGRATION OF OFF-THE-SHELF TOOLS	DEVELOPMENT OF CUSTOM COMPONENTS
DEVELOPMENT OF SPECIFICATION LANGUAGES	USE OF ENHANCED SAEF IN PRODUCTION	CONTINUATION OF THE TOOL INTEGRATION
DEVELOPMENT OF ANALYSIS TECHNIQUES	INTEGRATION WITH SAEF CAPABILITIES
		PRODUCTION EXPERIENCE
		TSD FACILITY REVIEW AND PLANNING

Figure 4-2. TSD FACILITY DEVELOPMENT MASTER PLAN

4.5 TSD FACILITY AND SYSTEM DESIGN AT DMA

DMA is in a position to take advantage of the TSD technology in several important ways:

- Contractors could make use of the envisioned TSD Facility on projects involved in the development of DMA systems;
- The TSD technology could be used by DMA contractors, even in the absence of the TSD Facility, particularly in the design of systems which are distributed in nature and involve decisions regarding the selection of a proper hardware/software mix;
- The core tools being developed for the TSD Facility are also needed as part of the DMA modern programming environment (MPE) which is seen as evolving in a TSD Facility specialized in software development;
- The TSD Methodologies may also be used in DMA on certain projects where the relation between software and hardware is important (e.g., the placement of various functions on a locally distributed system) and, thus, could affect DMA software development practices.

Figure 4-3 outlines a plan dealing with the last three of the four concerns expressed above. The direction being suggested here is analogous to that part of the master plan that deals with the refinement of TSD Methodologies. The distinction is not in the basic approach but in the scope and objectives. In the master plan the intent is to define the scope of and to support the long range R&D efforts in the area of distributed system design. Here, the objective is technology transfer from the R&D domain to actual production for the sake of achieving immediate quality and productivity improvements. As such, the emphasis is not on developing novel design, specification, analysis, and other techniques but rather on adapting already existing techniques for use in some particular application in a manner compatible with the TSD philosophy. It is conceivable that after empirical evaluations via appropriate pilot projects, some limited use of the methodologies on selected projects will become feasible in the near future. The potential impact of such endeavors on the DMA modern programming environment, on its approach to system development, and even on its software development standards should not be underestimated.

The results of this kind of highly pragmatic investigation could be instrumental in disseminating the TSD technology and its benefits to organizations which need it, in promoting tool development efforts which would later contribute to the evolution of TSD Facilities, and in stimulating more rapid exchange of ideas between researchers and practitioners in the field of distributed system design. Furthermore, this secondary plan is fully consistent with the TSD Facility master plan and it is necessary in order to assure broad application area coverage when practical considerations impose severe limitations on the scope of the proposed TSD Facility.

IDENTIFICATION OF POTENTIALLY
HIGH PAYOFF AREAS AS
CONTRACTOR AND AS DEVELOPER

REFINEMENT OF THE TSD
METHODOLOGIES WITH RESPECT
TO THE SELECTED AREAS

EMPIRICAL EVALUATION OF
THE METHODOLOGIES ON SEVERAL
SMALL PILOT PROJECTS

EVALUATION OF THE DMA
MODERN PROGRAMMING ENVIRONMENT
WITH RESPECT TO ITS ABILITY
TO SUPPORT THE METHODOLOGIES

ENHANCEMENT OF CURRENT DMA
ENVIRONMENT TOWARD BEING
BETTER PREPARED TO RESPOND
TO FUTURE SYSTEM DESIGN NEEDS

LIMITED USE OF TSD METHODOLOGIES
ON SELECTED DMA PROJECTS

REEVALUATION OF SYSTEM DESIGN
NEEDS AND AVAILABLE TECHNOLOGY
AT DMA

Figure 4-3. DMA OPPORTUNITIES FOR USE OF TSD TECHNOLOGY

APPENDIX A

ANNOTATED BIBLIOGRAPHY
(PERTINENT GOVERNMENT REPORTS)

Authors: David A. Bennett, Christopher A. Landauer, Mark E. Radlowski

Title: Automatic Integration of Multiple Element Radar

Source: PAR Corporation, under RADC Contract F30602-78-C-0139

Abstract: A case study of an application of the SAEF facility at RADC is presented, along with a complete evaluation of the facility. The application is in the domain of Command Control and Communication (C3); the development of a radar system, employing several independent radars to track multiple ground vehicle targets. The SAEF facility was employed in the emulation of a network of loosely coupled, distributed PDP-11 - type processors. A variation of the PDP-11/70 CPU was developed to emulate the separate processors, termed a PDQ-11 since it ran on the Nanodata QM-1. The PDQ-11 emulators were developed in SMITE and implemented in MULTI, the QM-1's microassembly language.

The report presents an overview of the tracking application logic, describes the components of the AIMER system emulation, discusses problems with the SAEF facility, and makes several recommendations for the improvement of the facility. Progress on the AIMER project was impeded primarily because of faulty or incomplete documentation of the SAEF facility and problems with the interface between the QM-1 and the MULTICS support system. PAR recommends research into the development of a coherent, well-integrated software development system (perhaps UNIX) to be used in conjunction with the QM-1.

Authors: Donald Boyd, Antonio Pizzarelli, Stanley C. Vestal

Title: Rational Design Methodology

Source: RADC Technical Report RADC-TR-78-208

Abstract: This report describes an effort to specify a software design methodology applicable to the Air Force software environment. Available methodologies and techniques were examined and investigated for (1) level of completeness; (2) ability to conform to Air Force design practices; and (3) inclusion of techniques for proof of correctness, design specification, and performance assessment of static designs. The rational methodology selected is a synthesis of ideas including data abstraction and refinement, constructive approach for software design, documentation procedures and tools. As a demonstration, the methodology was used to design a major function in the IBM Program Support Library.

Authors: Benjamin Britt, Alvin Cooperband, Louis Gallenson, Joel Goldberg

Title: PRIM System: Overview

Source: Information Sciences Institute
ARPA Report ISI/RR-77-58

Abstract: This document is an introduction to the services available with the Programming Research Instrument (PRIM), an interactive microprogrammable environment available to remote users through the ARPANET. PRIM, which runs under the TENEX timesharing system, makes it possible to create and use emulators of existing or newly specified computers, with major emphasis on debugging tools. PRIM and TENEX together provide not only editors, compilers, and debuggers for creating emulators, but also an environment for using the target systems, debuggers, and configurers in the familiar language of the original system.

Author: Capt. N. Bruce Clark

Title: Common Software Support Environment

Source: White paper, RADC

Abstract: This paper describes the costs associated with the traditional practice of providing an independent support system for each weapon system embedded computer system (ECS). These costs include the physical plant associated with a hotbench configuration, the software tools needed to prepare, debug, and evaluate software for that ECS, and the training and staffing of associated personnel. It is observed that the unique aspects of a hotbench require its continuation. However, the support software and support personnel could be provided at a separate facility. This would have a comprehensive set of software tools such as editors, assemblers, and simulators. In addition, it is suggested that the facility have an emulation capability that could support software testing tools that are not commonly supportable in a hotbench environment. The emulation capability would also enable proposed hardware designs (or modifications) to be evaluated for proper function and performance prior to procurement. The paper concludes by describing the efforts toward the implementation of such a support facility being carried out by the Rome Air Development Center (RADC), Rome, New York.

Author: Capt. N. Bruce Clark

Title: The Total System Design Methodology

Source: White paper, RADC

Abstract: This paper points out that the cost and performance of a system can be adversely affected by deciding too early in the process of system development on the hardware to be used. In particular, since hardware costs are known to be a small part of total development costs, the hardware decision should be tailored to the needs of the system and therefore not frozen until most system details have been worked out. This paper presents a development methodology which emphasizes the dependent role of hardware. This methodology, called the Total System Development (TSD) Methodology, requires the following resources: a set of languages for formally representing the system design at various stages in the development process; a set of software tools for managing the development process and for automating many of the development tasks; and an emulation facility to allow the system (or parts thereof) to be evaluated for functional and performance acceptability. Efforts by RADC to acquire the appropriate resources are discussed. These include an emulation facility called the System Architecture Evaluation Facility (SAEF), being built at RADC, and various language and software products being developed under RADC sponsorship.

Authors: Capt. N. Bruce Clark, 2Lt. Michael A. Troutman, USAF

Title: The System Architecture Evaluation Facility, an Emulation Facility at Rome Air Development Center

Source: White Paper, RADC

Abstract: The System Architecture Evaluation Facility (SAEF) is designed to provide an experimentation laboratory for research into advanced hardware configurations necessary to support the complex data processing needs of military command, control and communications systems. Elements of SAEF include a Nanodata QM-1 as the primary emulation tool, and a DECsystem-20 with Q-PRIM (an interactive microprogramming environment). A Multiple Microprocessor System (MMS) is currently in the design stage and, when completed, will provide the capability to emulate a wide variety of multiple processor architectures. Support tools include SMITE, which is a hardware description language which allows machine descriptions without resorting to microprogramming, and a retargetable compiler

to be developed for use with emulated architectures. Both basic research on computer architectures and systems development activities using the "Software First" concept are possible with the SAEF.

Authors: Jack M. Dreyfus, Peter J. Karacsony

Title: The Preliminary Design as a Key to Successful Software Development

Source: TRW Defense and Space Systems Group Report TRW-SS-76-09

Abstract: The paper predicates the success of a software development effort upon the establishment of a complete preliminary design emphasizing: (1) clear definition of data processing requirements, (2) top-down design definition, (3) design traceability, and (4) design verification. The report describes TRW's preliminary design methodology which has successfully been applied to large scale software development. The Methodology is a disciplined integration of design activities from initial system definition to successful completion of the software Preliminary Design Review. The benefits from complete preliminary design and some of the specific design techniques employed are described.

Author: Harris Corporation GCS Division

Title: Multiple Microprocessor System (MMS) Design Study

Source: Rome Air Development Center
Technical Report RADC-TR-80-33
(RADC Contract F306602-78-C-0114)

Abstract: The report describes and justifies the design of the Multiple Microprocessor System (MMS). A major component of the System Architecture Evaluation Facility (SAEF), MMS is intended for use in the emulation and evaluation of a wide range of multi-processor configurations. The proposed hardware consists of 64 processing units and several other specialized control and monitoring components. The communications take place via a segmented bus. The design choice is justified by means of a statistical analysis based on expected characteristics of the systems to be modelled. The hardware design is followed by the specification of the companion software needs for the MMS.

Authors: Charles Hayden, Peter W. Alvin, Stephen D. Crocker
Title: Multi-Microprocessor Emulation Annual Report for 1977
Source: Information Sciences Institute
ARPA Report ISI/SR-78-12
Abstract: The goal of the Multi-Microprocessor Emulation (MMPE) project is to develop modeling and emulation techniques for assemblies of microprocessors. An extension to an existing computer description language (ISPS) is proposed for representing the architecture of multi-microprocessor systems, and the results of some preliminary studies on the design of an emulation facility are described. This effort will eventually lead to a high-speed emulation facility based on the Q-PRIM system. The emulation facility is one component of the SAEF under development at RADC.

Editors: Raymond C. Houghton, Karen A. Oakley
Title: NBS Software Tools Database
Source: National Bureau of Standards
Report NBSIR 80-2159
Abstract: The paper contains a compilation of data on the availability of software development and testing tools. The data that has been compiled has been placed into a relational database using Pascal/R, a language that extends Pascal by a data relation. The database allows for information retrieval on tool features, languages, developers, documentation, hardware and software requirements, availability, publications, and contacts. The purpose of the report is to put forward the information currently contained in the database for review, assimilation, and update. Section 2 contains the Call for Tools. This section includes instructions for providing information on tools for inclusion in the database. Section 3 is a discussion of the records that may be stored in the database. Section 4 is a dump of the information contained in the database in alphabetical order by tool acronym. Section 5 is a list of general references. The appendices include several cross-references to the tools in the database and a list of specific tool references.

Author: Raymond A. Liuzzi

Title: The Specification of a Data Base Machine Architecture Development Facility and a Methodology for Developing Special Purpose Function Architectures

Source: Rome Air Development Center
Technical Report RADC-TR-80-256

Abstract: This report specifies the set of components/tools needed in a Data Base Machine Architecture Development (DMAD) Facility. A methodology is described to illustrate how this proposed facility can be used to develop special purpose function architectures (SPFAs). These SPFAs perform a data base management function in hardware that is currently performed in software on a sequential computer. The methodology includes a series of processes which are the select candidate function process, and the create, test, evaluate, and substitute SPFA processes. Each process can be performed with a series of procedures that utilize tools/components of the DMAD Facility. Tests and measurements conducted in order to illustrate the feasibility of generating detailed analysis data prior to any actual hardware implementation of a SPFA are also presented. This type of data is shown to be invaluable in helping project the highest qualified SPFA candidates to actually be hardware prototypes, and in providing input that can be used in their actual hardware implementation.

Author: Martin-Marietta Aerospace

Title: Total System Design Methodology

Source: Martin-Marietta Technical Report MCR-79-646

Abstract: During the last several years the experience with complex command, communication, and control (C-cubed or C3) systems has indicated that a systematic, rational approach to computer systems design is needed. Martin-Marietta has produced a Total System Design Methodology to support such design. This methodology includes both a philosophy of design and a framework for carrying out a design, along with some automated tools to aid in information gathering and ordering. The purpose of the paper is to document the existing TSD methodology at Martin-Marietta, describe the supporting tools, and review the use of the methodology on the design of the Navstar Global Positioning System Operational Control Segment.

Author: Gruia-Catalin Roman

Title: A Methodological Framework for the Design of Distributed Systems

Source: Washington University Technical Report WUCS-79-10 (RADC Contract F30602-78-C-0148)

Abstract: Building on the fundamental assumption that effective methodologies are problem and environment dependent, a suggestion is made to distinguish between methodologies and the methodological frameworks they instantiate. TSD (Total System Development) is put forth as a candidate framework able to assist in the generation and evaluation of specific system development methodologies, where systems are defined as distributed hardware/software aggregates.

Author: Rome Air Development Center

Title: Reconfigurable Computer System Design Facility Initial Design Study

Source: RADC Technical Report RADC-TR-78-6

Abstract: The total system design concept envisions a disciplined system design environment that allows overall system designs and alternatives to be quickly and easily evaluated, thus minimizing the actual development and life-cycle costs for new systems. A total system design facility is required to provide the necessary tools, evaluation techniques, and methods that support such an environment. The objective of the initial Reconfigurable Computer System Design Facility (RCSDF) design study was the preparation of a development plan describing the necessary studies and development tasks that would achieve the required facility capabilities. The initial RCSDF design study was organized into three major tasks: (1) Evaluation and definition of RCSDF capabilities, philosophy, and procedures; (2) Performance of RCSDF technical baseline development studies; and (3) Preparation of a RCSDF development plan. The three tasks of the initial RCSDF design study led to a development plan for a demonstration of the total system facility concept with available hardware and technology during the 1980s.

Author: TRW Defense and Space Systems Group
Title: SMITE Installation and Analysis - SMITE Training Manual
Source: TRW Report 30417-6002-RU-00
Abstract: This document is written for the person who wants to understand the use of the SMITE computer description language in the solution of problems of computer architecture and emulation in computer information systems development. The book has three distinct roles: (1) It is the primary material used in formal training classes on SMITE. (2) It is suitable for self-study use by persons familiar with basic computer architecture and emulation concepts. (3) It is suitable for use as a reference manual for users of the SMITE language, compiler, and associated support software.

Author: TRW Defense and Space Systems Group
Title: FAST Methodology and Case Study
Source: Final Report on Contract F30602-79-C-0078, RADC
Abstract: The overall presentation is organized top-down, from general to particular, over several levels of abstraction. First, the entire system development is considered from a highly abstract nonprocedural point of view in order to identify the role of hardware/software trade-offs and the way in which they relate to other system development activities. Second, issues pertinent to the functional and performance specification of systems design are reviewed. The report next focuses on the fundamentals of hardware/software tradeoffs and proposes a general approach to carrying out the tradeoffs analysis. Finally, application of this method to a DMA based case study is described in detail.

APPENDIX B
GLOSSARY OF TERMS

LOGICAL GROUPING OF TERMS

framework
methodology
facility
stage
phase
step

TSD
TSD concept
TSD framework
TSD methodology
TSD facility

problem definition stage
system design stage
software design stage
machine design stage
circuit design stage
firmware design stage

identification phase
conceptualization phase

system architecture phase
system binding phase

software configuration design phase
program design phase
coding phase

hardware configuration design phase
component design phase

switching circuit design phase
electrical circuit design phase
solid state design phase
fabrication phase

microcode design phase
microporgramming phase
microcode generation phase

formalism selection step
formalism validation step
exploration step
elaboration step
consistency checking step
verification step
evaluation step
inference step
invocation step
integration step

life-cycle
development
analysis
enhancement
maintenance

performance

system
embedded computer system
C-cubed system
information processing system

requirements specification
specification language
conceptual model
processing model
constraints

H/S trade-offs

DEFINITIONS IN ALPHABETICAL ORDER

analysis

The process of assessing some performance property or properties of a system by examining it or its specifications.

C-cubed system

A computer system supporting the command, control, and communication functions within some military outfit.

circuit design stage

A stage of the TSD framework.

coding phase

A phase of the software design stage.

component design phase

A phase of the machine design stage.

conceptualization phase

A phase of the problem definition stage.

conceptual model

A model formalizing an application problem in terms of abstract application domain concepts and independent of possible system realizations. It is produced by the conceptualization phase.

consistency checking step

A step in the unified phase structure.

constraints

Factors limiting the domain of acceptable design solutions. They may originate with the customer, technology, rules of the trade, previous design decisions, etc.

development

The set of all activities involved in the generation of a first version of some system, from initial concept to production and deployment.

elaboration step

A step in the unified phase structure.

electrical circuit design phase

A phase of the circuit design stage.

embedded computer system

A computer system supporting real-time process control functions, such as flight control, firing control, guidance, etc.

enhancement

The process of modifying an existing system in order to acquire additional or different functional/performance characteristics.

evaluation step

A step in the unified phase structure.

exploration step

A step in the unified phase structure.

fabrication phase

A phase of the circuit design stage.

facility

The resources available at some location for use in the application of methodologies to various problems.

firmware design stage

A stage of the TSD framework.

formalism selection step

A step in the unified phase structure.

formalism validation step

A step in the unified phase structure.

framework

A high level non-procedural description of some general problem solving approach which identifies: (1) a set of subproblems whose solutions lead to solving the target problem, and (2) the fundamental relationships among subproblems without regard to the manner in which one arrives at their solution.

H/S trade-offs

Hardware/software tradeoffs. The process of, and issues involved in, deciding the assignment of a system's functions to various types of physical system components.

hardware configuration design phase

A phase of the machine design stage.

identification phase

A phase of the problem definition stage.

inference step

A step in the unified phase structure.

information processing system

A computer system supporting some application area (business, project management, logistics command) by enabling the acquisition, storage, and retrieval of pertinent information.

integration step

A step in the unified phase structure.

invocation step

A step in the unified phase structure.

life-cycle

This denotes the period of time from the conception to the retirement of a system, as well as all activities involving the system, its development, analysis, enhancement, and maintenance.

machine design stage

A stage of the TSD framework.

maintenance

The process of (1) modifying an existing system in order to correct deviations from its functional/performance specifications, and (2) of replacing obsolete or defective components.

methodology

A mode of procedure to be followed in solving a given problem. It exploits particular features of the problem and environment through the use of specific techniques or classes of techniques.

microcode design phase

A phase of the firmware design stage.

microcode generation phase

A phase of the firmware design stage.

microprogramming phase

A phase of the firmware design stage.

performance

A collection of attributes associated with the structure or behavior (though not functionality) of a system (e.g., response time), or activities involved in a system's life-cycle (e.g., maintenance costs).

phase (of a methodological framework)

A design problem formulated as a transformation between two requirements specifications and involving activities within the same knowledge domain.

problem definition stage

A stage of the TSD framework.

processing model

A system design description produced by the system architecture design phase.

program design phase

A phase of the software design stage.

requirements specification

A consistent and complete description of some problem statement, or interim solution to a design problem, or some fraction thereof.

software configuration design phase

A phase of the software design stage.

software design stage

A stage of the TSD framework.

solid state design phase

A phase of the circuit design stage.

specification language

A language used to state a problem, or to describe the solution to a problem.

stage (of a methodological framework)

A hierarchical group of related phases.

step (of a methodological framework)

A subproblem fundamental to the solution of the problem identified by a given phase.

system

A hardware/software aggregate.

system architecture design phase

A phase of the system design stage.

system binding phase

A phase of the system design stage.

system design stage

A stage of the TSD framework.

switching circuit design phase

A phase of the circuit design stage.

TSD

An acronym standing for Total System Design.

TSD concept

A viewpoint which envisions system design as taking place in a support environment consisting of a family of design methodologies and a collection of associated design aids. Moreover, the TSD concept also presumes the ability to explore easily the space of design alternatives every step on the way, and to take rational decisions based primarily on solid technical reasons. The notion of avoiding premature commitments to particular design solutions, such as the a priori selection of specific hardware, is another key component of the concept and one of the motivating factors behind its inception.

TSD facility

A facility providing support for the class of TSD methodologies.

TSD framework

The TSD framework is a methodological framework that forms the foundation of a class of system design methodologies whose goals are: (1) to recognize formally the H/S dualism, (2) to avoid premature hardware selection, (3) to minimize error costs through early error detection, (4) to treat performance constraints as a major driving force behind the design process, (5) to promote design automation, and (6) to enable proper evaluation of human interfaces.

TSD methodology

Any methodology compatible with the TSD framework definition.

verification step

A step in the unified phase structure.

APPENDIX C
THE TSD METHODOLOGY GUIDEBOOK

C.1 Overview

This Appendix offers a brief introduction to the ideas and attitudes surrounding the methodical design and development of systems. A "system," in this context, is defined as a computer-based set of hardware and software components for processing information in support of one or more applications. The application(s) for which the system is intended may require that system to exist as a physically distinct entity (i.e., a standalone system), or it may be an integral component of some larger complex (i.e., an embedded system). In any event, the design, development, implementation, use, and maintenance of such systems plays a significant role in DoD's activities.

Advances in the underlying technology, along with continuing demands from a growing range of sophisticated application areas, are causing dramatic increases in system complexity. As a result, the use of traditional (ad hoc) approaches to system development has become progressively less effective in meeting requirements for timeliness, reliability, and cost effectiveness. The situation is aggravated by a burgeoning microelectronics technology that has presented designers with unprecedented hardware alternatives. Opportunities to consider this broader range of choices in computer architecture in a given situation often go unexploited because current system design/development practices tend to be dominated by a "hardware first" philosophy in which software is superimposed on a hardware system whose characteristics are completely defined early in the system life cycle - even before the functional requirements are completely clear.

These factors have prompted a growing interest in (and movement toward) more systematic design methodologies in which engineering principles used effectively for general product design/development are being applied to computer-based applications. As a result, numerous methodologies have emerged in an effort to impose more discipline on the system design process, and a variety of tools have become available in support of these methodologies. The effectiveness of this systematization has been demonstrated in a wide range of application areas, so that these methodical approaches are rapidly replacing traditional ways.

With the increasing use of orderly system design methodologies has come a growing awareness that the effectiveness of a given methodology may depend strongly on the type of application being developed. DoD, being one of the first organizations to recognize this dependency, saw the need for a way to deal with multiple methodologies and the assessment of their applicability to DoD needs. The result is the Total System Design (TSD) concept, a logical framework within which system design methodologies (and tools used as components of such methodologies) can be organized and considered.

Within the perspective outlined above, this Guidebook seeks to:

-- acquaint its readers with the major benefits derived from the use of orderly methodologies in the system design process. Toward this end, Section C.2 briefly traces the factors underlying the development of an acknowledged

software crisis and its broadening growth to a system crisis. Response to the symptoms of that crisis is characterized as a continuing shift away from ad hoc approaches toward a more systematic orientation in which principles of engineering project design, management, and control play an increasingly important role. These developments are related to the problems they are intended to relieve. Introduction of technological advances, especially those in the microelectronics area, establish the need to accelerate the changing perception of the system design process so that hardware architecture is included as a design variable. At the same time, proliferation of design methodologies and tools is shown to prompt a need for a general framework within which these resources can be examined. This sets the stage for the TSD framework.

- define the TSD concept and establish its framework as a vehicle for classifying, analyzing, and comparing system design methodologies. Accordingly, Section C.3 characterizes the framework as a logical structure in which the components of the system design process are abstracted, categorized, and organized into a cohesive whole. The entire process is divided into stages, and each stage's duties are described in terms of major activities called phases. Each phase, in turn, is comprised of ten standardized steps required to bring that phase's work to fruition. The importance of any step is seen to depend on the phase in which it is being considered while the importance of any phase or stage is seen to be dictated by the particular application being examined within the framework.
- acquaint the reader with the nature and extent of tools that are currently available in support of the system design effort. This is done in Section C.4 where each phase in each step of the TSD framework is associated with the kinds of available resources that aid in performing aspects of that phase's work. While some tools relate uniquely to a particular phase (for example, a compiler for a high level programming language is applicable specifically to the coding phase of the software design stage), others (like a text editor or report generator) may help support the work in each phase of several stages. Reference also is made to the growing practice of combining tools implemented for a given computer system and integrating them within appropriate supervisory software to form a computer-based working environment for design, programming, project management, or some other major activity in the system cycle.
- characterize the nature and direction of expected future developments in system design methodologies. Although extrapolation always involves some speculation, it

appears highly likely that there will be continued movement from individual tools and methodologies to computer-based design/development/management environments offering a range of methodologies (and opportunities to build new ones). A second contention is that continuing demands for increasingly complex systems will obligate more and more organizations to embrace a design philosophy in which *a priori* hardware selection is no longer acceptable as a universal rule of practice. Instead, system duties will be relegated to hardware/firmware or software as the result of a design activity in which hardware/software tradeoffs are seriously examined. These two ideas are combined in Section C.5 to form the basis for projections that envision growing acceptance of a total system design approach supported by methodologies consistent with the TSD framework. Such methodologies, in turn, are expected to be made available on increasingly versatile facilities that include hardware emulation capabilities and are configured expressly for support of system design activities across the entire life cycle.

C.2 Background

The application of orderly, disciplined methodologies to computer-based system design/development is an idea whose general acceptance reaches into every area in which computers are involved. In fact, the precepts of software engineering have been institutionalized to a sufficient extent that many practitioners find it increasingly difficult to imagine an alternative. Yet, this seemingly inevitable approach lagged the introduction of computers by almost two decades. During this initial period, little or no attention was paid to systematization; ad hoc development prevailed. As Clark points out [CLAR79a], this approach to computer applications development was characterized further by a perception that emphasized early selection and procurement of hardware. Consequently, software development was a process which started with the intended hardware already defined. This "hardware first" approach persisted throughout the computer community even after software costs began to dominate overall system costs. The need to establish system requirements as the driving force for both hardware and software definition now has begun to gain recognition. An important response to this need has been DoD's Total System Design (TSD) methodology. This section examines the shift toward a TSD approach by discussing the forces that brought it on.

C.2.1 Introduction

In 1961, a computer equipped with 8000 bytes of main storage, a card reader/punch, and a line printer was considered a medium-sized constellation. Such a system cost two hundred forty five thousand dollars; 1961 dollars. Its memory speed was about five percent of that seen in today's personal computers. This is pointed out to underscore the fact that the hardware was the predominant financial factor in most computer installations. The resulting effect on computer usage and its management was profound. Moreover, its consequences continue to persist even though the financial factors have changed drastically. The climate produced by this situation can be characterized briefly as follows:

1. Productive computer utilization was uppermost. Managers were eager to justify sizable hardware expenditures by filling the available machine time with useful computer applications. For many installations it was typical for the equipment to be acquired initially for certain applications. Once implemented, these applications consumed a relatively modest fraction of the time. In spite of the fact that procurement of the computer often could have been defended solely on the basis of these initial applications, "idle time" was something actively to be minimized. While this certainly was not the only factor, it did contribute significantly to the rapid growth in the number and diversity of computer applications during the Fifties and well into the Sixties. From another point of view, there was an even more telling effect: The drive to fill a computer's available time helped establish and reinforce a tradition in which a computer application is perceived in terms of a program (or a complex of interrelated programs) written for a computer whose

configuration and architecture are "given". That is, the hardware is (essentially) defined before the application is conceived. Advances in microelectronics have made this operating mode increasingly obsolete for many types of applications. By the same token, design approaches predicated solely on this basis can impose unnecessary constraints. Consequently, when this tradition is broken, a significant underlying concept emerges in which the hardware and software are treated as parallel design issues, both driven by the application. This concept, which we term the hardware/software duality, is one of the focal points behind the TSD framework.

2. Computer applications were considered to be relatively stable. Many of the early computer applications were transferrals of procedures done previously by hand or with unit record equipment. These generally were established processes whose requirements were well understood. It is not surprising, therefore, to find the same kind of stability being attributed to applications with no prior counterparts. This perception often turned out to be erroneous. For instance, requirements defined at some early point in a system's life cycle were seen to be inadequate or improper because of information that came to light during subsequent development. Alternatively, systems deemed at first to be satisfactory tended to lose their appeal as experience with them accumulated: Operating features that were unanticipated at first were recognized later on as being desirable or even essential. Consequently, there was a pervasive and growing discrepancy between the perception of stable applications and their actual dynamic nature. This idealistic view was to be a crucial factor in precipitating what turned out to be nothing less than a software revolution.
3. Program efficiency was a primary software design objective. Although such issues as software maintenance and reliability were recognized and considered, attention in software development focused primarily on the production of programs in which size and execution time were minimized. Main storage and computer time both were perceived (and treated) as precious commodities, so that their conservation was a matter of high priority. Such emphasis was not misplaced. Restrictions imposed by the available hardware often forced the development of applications that operated at or near system capacity. This meant that programmers tended to accumulate shortcuts and special tricks which saved words of storage or microseconds of execution time. By and large, these techniques were not algorithmic; rather, their effectiveness was based on specific idiosyncrasies intrinsic to the particular computer, language, or operating system being used to implement the application. The resulting changes in a program usually helped obscure its intent. Thus, it was not unusual for situations to develop in which a programmer, looking at the listing of a program he or she had written some weeks earlier, could not discern what that program did or how it did it.

A more fundamental factor contributed to the perceived pre-eminence of program efficiency: software design and programming were treated as an integral activity, with neither conceptual nor temporal distinctions being made between them. Consequently, it was inevitable to find system and program efficiency being intermixed.

4. Error removal was associated with the latter stages of the software development cycle and was handled predominantly by program debugging. For many practitioners at every level of computer applications development, the programming process was defined rather vaguely. It was seen to include aspects of algorithm design, requirements definition, and optimization in addition to the actual production of coded programming language statements. Understandably, then, it was expected (and accepted) that an initial version of a program (or subprogram) would contain a mixture of errors attributable to any or all of these factors and not just to syntactic/semantic misstatements vis-a-vis the rules of the programming language. Ultimately, all of these types of difficulties would be sought, discovered, and corrected when the completed program is debugged. As the discussion in the next two sections makes clear, the adverse effects of this orientation cannot be overstated.

C.2.2 Current Problems and Concerns

The difficulties encountered in the design, development and implementation of computer-based systems are deceptively easy to characterize: Compared to earlier systems, more recent ones have tended to:

1. exhibit greater discrepancies between estimated and actual costs, with software costs assuming an increasing fraction of the total.
2. suffer greater time delays in their preparation.
3. contain more errors when released for use, so that maintenance (correction of errors to make a system match current requirements) is becoming an increasingly significant cost component.
4. be more difficult to enhance properly in response to changing requirements.

There is no intent to imply that early systems were free of such troubles. However, trends have been observed wherein these tendencies have been intensifying with time. In a widely quoted landmark study that includes many DoD projects [BOEH73], software is seen to have become the dominant cost component (as hardware price/performance figures decrease and labor costs continue to go up), with maintenance/enhancement constituting the primary ingredient in software cost. Similar findings are reported in [CLAR79b] and [FASP].

There is little doubt that these trends can legitimately be attributed to the continuing growth in the complexity of more recent computer applications and the systems required to implement them. However, if this trend is to be arrested and, ultimately, reversed, it is necessary to understand why the increase in complexity should exert such an adverse influence.

Severe problems in the design and development of computer applications did not arise overnight. When they did appear, they generally were unanticipated and often misinterpreted. It would be misleading to think of this as shortsightedness. The simple fact is that innumerable projects and applications were implemented successfully, with performance being improved dramatically over previous versions in which computers were not involved. Failures, viewed in their own context, were thought to be due perhaps to an unrealistic time constraint, too few programmers, or inadequate programmer quality. Thus, there was little or no impetus to examine the programming process or the broader scope of activities related to the preparation of a system. At the time "third generation" computers began to emerge from the production lines, programming was still treated as a craft taught by masters to apprentices, and the design and realization of computer-based systems continued on an ad hoc basis.

The potential for severe software problems already was present with the first computer applications. However, the recognition of the problem as being intrinsic did not spread until newer systems became sufficiently complex to begin stressing the capabilities of ad hoc system design approaches beyond their limits. Experience with large software projects [BRO075] shows that as projects increase in scope and more programmers are assigned to work on them, the positive effect of the additional people tends to be neutralized and eventually overwhelmed by the rapidly increasing complexity of the required communication among all of the cooks working on the various parts of the same broth. As long as the projects were limited, this clash of effects was not apparent. However, with increasing growth, projects became more vulnerable to failure through misinterpretation, duplication of effort, lost information, and other consequences stemming from inadequate coordination. To counteract this susceptibility, it was necessary to devote an increasing (and ultimately disproportionate) fraction of personnel time and effort to making sure that all concerned parties knew what they needed to know. By the mid-Sixties, many organizations were undertaking system projects of sufficient complexity to bring these problems into prominence. This was particularly true in DoD, where C-Cubed systems and embedded computer systems were assuming an increasing role in the Department's activities.

This increase in complexity is a natural phenomenon. There is hardly a human endeavor that is exempt from the forces of discontent pushing toward "improvement" and "expansion". Something (a computer-based system, in our context) that is deemed "effective" or even "outstanding" when it first appears, soon becomes "adequate" and, ultimately, "mediocre" or even "unsatisfactory". In addition, requirements motivating a particular system often are imposed by external sources so that arbitrary changes in requirements (leading to complications more often than simplifications) may appear at arbitrary points in the system's history. (DoD systems are

no exception.) Thus, if nothing else were to happen, it still would have been inevitable for software problems to surface as endemic characteristics of the development processes then current.

As it turns out, the emergence of an acknowledged "software problem" has been accelerated (indirectly) by a rush of technological advances. On repeated occasions, new, less expensive hardware with higher capacity, increased speed, and more versatile configurational possibilities has engendered dissatisfaction with application systems previously viewed as being adventurous. Similar effects have been (and continue to be) wrought by more powerful languages and operating system software as well.

The effect of these advances in computer science and technology on application systems is even more fundamental than that just mentioned. Briefly stated, the availability of a wide variety of powerful, fast, inexpensive processors has expanded what was perceived as a "software problem" to a "hardware/software problem". Fulfillment of a particular set of requirements can no longer be viewed solely in terms of a traditional "hardware first" solution in which a software structure is superimposed on a predefined hardware architecture. Thus, hardware architecture has become a legitimate design variable, to be considered on a par with software issues(*). Failure to exploit these expanded hardware possibilities often contributes to the severity of the overall systems problems. In the next section, when responses to these problems are discussed, we shall examine conceptual mechanisms that allow for the natural inclusion of hardware considerations as an integral part of a total systems design.

C.2.3 Response to Increasing Complexity

Any reasonable attempt to relieve the problems cited in the previous section must stem from the realization that the phenomenon of increasing complexity will not go away. As we continue to learn more about the behavior of current systems, our expectations grow accordingly, and they manifest themselves as more ambitious demands and challenging requirements for future systems. These new systems, expectedly, will turn out to be more complex than their predecessors.

Faced with this reality, attacks on systems problems are based on the idea of reducing apparent complexity. This is the common objective that motivates all efforts to systematize the design, implementation, and maintenance of a computer-based application. For any aspect of that process (such as software design or programming), the intent is to represent a pertinent problem as a collection of interrelated but distinct subproblems, each one sized so that an individual can deal with its entire scope and all its details. Such subdivision is effective when an individual working on a particular subproblem can focus full attention on it with minimal concern about how it relates to the overall system.

* Perhaps the most significant characteristic of so-called "third generation" computer systems is the concurrent design of their hardware and executive software.

Later, when all of the (temporarily) isolated subproblems have been resolved, they can be brought together and integrated to form the final product. Since this orientation closely resembles the classical approach to product engineering, the term "divide and conquer" often is used to characterize methodological computer-based system development as well.

Even from this brief characterization, it is evident that the success of such an approach requires careful definition of the subproblems and the connections among them. Consequently, considerable work has been directed toward devising useful methods for establishing and documenting such definitions, facilitating their implementation, and enforcing adherence to them. Before such work could proceed fruitfully, it was necessary to deemphasize (or abandon entirely) some of the precepts underlying the traditional (ad hoc) approach to system design and development and replace them with a revised perspective more hospitable to systematic approaches. This need, not immediately apparent, was established by studying the programming process [WEIN70] along with other aspects of software design and development [YOUR75]. The salient features can be examined conveniently by contrasting them with their earlier counterparts:

1. Successful fulfillment of a computer-based system's requirements is perceived as a solution to a set of hardware/software problems. Advances in microcircuitry, improved manufacturing methods, and a deeper understanding of computer architecture have provided the system designer with an unprecedented range of hardware alternatives. The cost of this equipment has declined well beyond the point where idle computer time need be a matter of primary concern. Newly emerging technology in very large scale integrated circuits (VLSI) promises to reduce this concern even further [MEAD80]. This makes it possible (and practical) to derive the hardware requirements from those imposed by the application (rather than the other way around). The resulting configuration still may turn out to be a general purpose machine whose resources will be shared by several applications. However, the inclusion of hardware as a design variable introduces the opportunity to define equipment with specific architectural characteristics when the situation dictates it. Thus, instead of moving from the tradition "hardware first" perception of the system design process to a "software first" approach, increased architectural opportunities are prompting a shift to a "system first" philosophy in which both hardware and software can contend for selection as solutions to system component needs. As will be seen later, the family of TSD methodologies formalizes this opportunity.

Besides making processor utilization less prominent as a major objective, architectural flexibility introduces a more basic consequence: Many of the technological advances have blurred the distinction between those processing activities traditionally relegated to hardware and those automatically associated with software. For a growing number of processing activities, this choice no longer is clear-cut. As a result, more recent efforts to reduce apparent complexity include mechanisms that obligate the designer to examine a broader range

of possibilities so that hardware/software tradeoffs can be recognized and assessed.

2. Change is recognized as an intrinsic property of computer applications. Although this seems to be saying that water is wet, it does represent a significant change in orientation. The point is that traditional approaches to system design either assumed a stabilized set of requirements or resigned themselves to the inevitability of change with the implicit intention of accommodating changes (or fighting them off) when they came. A revised perspective accepts changing requirements as a basic system characteristic. Adaptability to change (enhanceability), then, becomes an explicit primary design objective to be met rather than a fortuitous byproduct when and if it happens. Inclusion of this criterion provides further motivation for careful subdivision of a problem into distinct but interrelated subproblems: with such decomposition, eventual enhancement of the system in response to changing requirements can proceed more smoothly if the alterations are localized to a small number of components. Consequently, the reduction of apparent complexity and the improvement of enhanceability are mutually supportive aspects of methodological system design.
3. Program clarity, simplicity, and reliability have emerged as prominent implementation objectives while efficiency has become an explicit concern that extends beyond the program. This stems directly from the separation that now has been established between design and programming/coding. Although program efficiency remains a serious concern, it is no longer the overriding factor to which all others are subordinated. Improvements in hardware, along with reduced costs, now make it largely unnecessary to develop systems that operate disturbingly close to equipment capacity. At the same time, accelerating increases in labor costs exert additional pressure against the (once traditional) use of skilled personnel to ferret out arbitrary (and often marginal) savings in execution time or memory use.

Efficiency, per se, is not seen as a primary driving force at the programming level. Instead, it is viewed in the context of the application being considered. This removes it from its earlier role as a programming/coding issue and expands it to one that is pertinent during all phases of the system cycle. As a result, there are numerous opportunities to include efficiency considerations on a continuing basis as a system takes conceptual, logical, and then physical shape. For example, if speed of execution is an important requirement (as it is in DoD's C-cubed applications), it can exert considerable influence on hardware choices and, more fundamentally, on hardware/software preferences. Similarly, it may affect the design of crucial algorithms. This latter effect is becoming increasingly important as theoretical advances continue to improve designers' ability to predict computational performance. As a result of these expanded opportunities, it is likely that the system design

already will be intrinsically efficient before it reaches a point where its software is ready to be implemented.

Once efficiency of the ultimate program is removed as a pivotal design concern, it becomes more fruitful to focus attention (when it comes to programming) on the efficiency of the programming process. Toward this end, the importance of program clarity and simplicity as primary programming objectives has been demonstrated repeatedly ([YOUR75], [WIRT73]). All of the precepts and practices being advocated under the umbrella of "structured programming" share a common goal: to make it easier (i.e., less expensive and less time-consuming) to produce a program that is organizationally and logically simple, clear, and more convenient to use. This means that its intent and its major processing characteristics are readily discernable, thereby making it easier to analyze (and modify, when required) and less likely to contain errors once it is released for use. These considerations constitute a major influence on the design and structure of the DoD's Ada language [DOD79].

This does not mean that program efficiency is abandoned; far from it. However, it places such considerations in their proper context: Once a program has been written and is being evaluated, its implementers are in an ideal position to observe its performance and pinpoint sources of inefficiency. Appropriate improvements then can be made systematically, by streamlining those individual programs or modules causing the bottlenecks. The consequences are localized and the process is more easily managed. Of particular importance here is the idea that these improvements are brought about by changes in the implementation, not in the design. In effect, this constitutes something akin to fine-tuning. It is worth repeating that when the (partially developed) system arrives at its software implementation phase, its efficiency is part of its design. The effect of the actual code, then, is likely to be less profound than that originally ascribed to it.

4. Error detection and removal are seen as explicit activities that pervade the entire system development cycle. A cornerstone of any disciplined approach to system design and development is the recognition that each distinct phase in the process must include a concerted effort to establish (to whatever extent feasible) the validity of the work produced by that phase. For effective methodologies, this is not merely a wish. It imposes a managerial responsibility to define and implement enforcement mechanisms that use such validations as criteria for embarking on subsequent phases.

As a result, the role of the programming process is perceived as being more sharply focused as an implementation activity. Since each prior phase is capped by a validation activity, the expectation is that programming is concerned solely with the implementation of algorithms whose logical and procedural correctness already have been established. An error

in the program, when viewed in this context, represents a failure to convey the intent of an underlying algorithm rather than a flaw in the algorithm or in the design that motivated the algorithm.

This orientation has helped efforts to devise improved approaches and techniques that allow progress toward the ideal situation in which a program is free of errors at the point of its first test. One technique, that of structured programming, already has been mentioned. Its objective, i.e., the imposition of consistency on the way programs are coded, is being fulfilled and the benefits ([BAKE72], for example) are widely acknowledged. Another side of this effort pays attention to the environment in which programming takes place. While the production of code continues to be an individual endeavor, today's programmer works (conceptually) in less isolation than in the past. In an increasing number of organizations, including many with heavy involvement in DoD-related computing projects, the programming process is supported by a variety of "programmers' tools" whose primary purpose is to facilitate (and help manage) the activities involved in preparing, refining, documenting, and monitoring the progress of software under development [KERN81]. Such aids continue to be a subject of intensive research.

In addition, programmers operate in a more structured managerial environment that lends administrative support to their efforts. Part of the task of reducing apparent complexity entails the establishment and maintenance of orderly communication channels among a project's participants. Innovations such as chief programmer teams [BAKE72] are proving to be effective in this regard. Other aids [TEIC77] are used to provide ample opportunities for program review and validation.

The conceptual separation of programming from system design also has highlighted the importance of program testing as an organized activity. Even if the abovementioned ideal of an initially correct program were to be met routinely, it still would be necessary to demonstrate a program's validity prior to its release. Such demonstrations never have been easy to define satisfactorily, and the task grows more difficult as systems become more complex. While it is impossible to perform an exhaustive demonstration of the validity of a realistic system, a well-conceived test plan, devised during development and not as an afterthought, can provide a demonstration that is both reasonable and convincing. Accordingly, test definition (for components as well as for the integrated system) is viewed as a design activity and not something to do ad hoc. This increases the likelihood of producing a systematic evaluation that includes what are thought to be the "most typical" episodes of system behavior as well as those that exercise the system at its limits.

5. Maintainability is treated as an explicit system characteristic.
We have mentioned enhanceability as a recognized property that facilitates incorporation of changes to systems already in use.

A related but distinct consideration is maintainability. Since the complexity of modern systems rules out any prospect of exhaustive testing, we must deal with the potentially strong possibility of a system containing residual errors even after successful completion of a well-conceived testing effort. Such errors may show up early in the system's usage or they may not surface for years; some may not appear during the entire life of the system. In any event, a maintainable system is designed to help pinpoint the source of an error when one occurs. One important implication of this view is that it encourages designers to devise ways in which the system can assume more of the responsibility for distinguishing between "normal" and "abnormal" behavior and explicitly reporting instances of the latter. This philosophy, already well established for hardware, is beginning to make itself felt with regard to software as well. For hardware, this help consists of special diagnostic components included to enhance the system's ability to report (and in some cases correct or circumvent) physical malfunctions. For firmware/software, sensitivity to possible errors is heightened by additional programming that can be activated when a hitherto undiscovered logical malfunction emerges. This diagnostic programming provides helpful information by revealing procedural details that are "invisible" during normal system use.

6. Diversity is seen as being potentially counterproductive as system complexity increases. Although increased hardware/software opportunities must be exploited if effective systems are to be assured, it is now understood that the advantages of a widened spectrum of choices are quickly neutralized by arbitrary, uncontrolled diversity. Without some degree of standardization, design and development costs often are inflated unnecessarily by the inability to make use of earlier work that would have been relevant if it were not for its incompatibility. This negative effect is intensified by the additional overhead incurred in having to familiarize technical personnel with a broader array of hardware/software products and their use in application systems development.

Consequently, the system design community has been moving toward an orientation that seeks to introduce standardization without compromising versatility. A prominent example is seen in DoD's effort to supplant a multiplicity of programming languages with a single one (Ada) designed to provide a consistent programming vehicle while still offering the wide flexibility required for embedded computer systems. Similarly, there are strong tendencies within DoD and other organizations to define stable architectures for single processors and input/output subsystems wherever appropriate. In a sense, there is a compelling similarity between this movement toward standardization and the more general one that helped characterize the Industrial Revolution.

7. There is growing recognition that system design and development can be helped considerably by the introduction of automation into the process. The basic concept certainly is not new to computing: automation of the program development process began with the appearance of the first high level language processor in the mid-Fifties. Since that time, there has been a concerted effort to define facilities and tools that would help automate other aspects of the system cycle. Until recently, these aids have concentrated on facilitating software development and system documentation by providing a variety of software-based supports. (Further discussion of these aids appears in Section C.4.2.) Now, the emergence of hardware and software as parallel design variables (i.e., the "system first" philosophy referred to earlier) is prompting growing interest in similar aids for hardware design and evaluation. An idea that is particularly prominent in this rapidly emerging technological area is the Air Force's System Architecture Evaluation Facility (SAEF). This is examined in Section C.5.

These responses to rapidly growing demands for more complex hardware/software systems manifest themselves as methodologies for orderly conduct of the activities encompassed by the system cycle. Resulting improvements in workers' productivity and system effectiveness and economy have placed such applications of structured design/development principles beyond dispute.

C.3 The TSD Framework

Work on disciplined system design/development methods has been intensifying since the late Sixties. As a result, there is a growing collection of resources aimed at supporting various stages of the overall process. Some of these are rather narrow in scope, taking the form of aids that expedite a particular step in a design stage; others are more comprehensive, providing an orderly approach that covers an entire design stage. At present, diversity, perhaps more than any other attribute, tends to characterize these resources. First, there is no single, integrated methodology that encompasses the entire system cycle. Individual approaches deal with various segments of that cycle, and the scope of one methodology does not necessarily coincide or dovetail exactly with that of another one. Moreover, many of the methodologies reflect the perceptions and concerns of the particular application areas that motivated their development and/or the environments in which they were produced. Thus, for example, a particular approach that was used effectively in the design of realtime software for an airline reservation system may not be as appropriate when applied to a vehicle dispatching system.

It is this realization that signals a transition to what might be considered a "second generation" in system design disciplines: In addition to continued work on individual methodologies, there is growing impetus to organize this knowledge into some sort of continuum that will enable people to exploit available technology in the most effective way for their particular system needs. The TSD framework, described in this

section, has been formulated to provide this kind of vehicle.

C.3.1 Characteristics of the TSD Framework

The TSD framework is not a method for designing computer-based systems. Rather, it is an abstraction of a class of system design methodologies. As such, it provides an organized way of looking at (and comparing) alternative approaches for handling various aspects of the system life cycle. Use of the framework is expected to facilitate the task of selecting methodologies and combining them to form the most effective total system design approach for a given set of circumstances. As new methodologies are introduced, their characteristics can be expressed in accordance with the framework's structure, thereby allowing the range of available choices to grow consistently.

It is helpful to point out that the TSD framework is not intended to serve as an abstraction for all system design methodologies. The class of approaches it is designed to accommodate can be characterized in terms of six common concerns:

1. A formal recognition of the hardware/software duality.
2. An explicit commitment to arrange the system life cycle so that premature hardware selection is unnecessary.
3. Inclusion of activities aimed specifically at early error detection.
4. Treatment of performance constraints as a major influence on the design process throughout the cycle.
5. Promotion of design automation.
6. Serious (and substantive) concern with the interfaces between a system and its human users.

To provide a structural basis for the TSD framework, the system life cycle is characterized by dividing it into major activities called stages. Each stage is divided further into phases and each phase, in turn, consists of a number of steps. Certain of these activities will require less emphasis for some systems than for others. However, from the framework's point of view, each activity is included (in concept) in any system's development, even if performance of the activity is trivial.

Steps within a phase (or phases within a stage) are not meant to follow each other in any particular sequence. Nor does the beginning of a particular step, phase, or stage automatically imply the irrevocable conclusion of one preceding it. The framework, by being abstract, is non-procedural, so that temporal relations among component activities can be dictated by the characteristics of the individual methodologies and the requirements of the applications. This abstraction makes it particularly convenient to include consideration of hardware/software tradeoffs as an explicit activity.

C.3.2 Structure of the TSD Framework

We shall begin examination of the TSD framework by characterizing its major structural constituents. Once the basic dependency relations have been established, we shall return to each stage and phase for a closer look.

C.3.2.1 stages: The conceptual relations among these major design activities is summarized in Figure C.1. Progress through the design process is indicated by the downward arrows, each of which represents requirements specifications developed by a particular stage. These, in turn, define the problems to be solved in the next stage. The overall downward flow is that of a system design working its way through increasing refinements until it is completely specified.

The diagram in Figure C.1 also is meant to convey the overall flow of the system integration process. This is represented by the upward arrows, each of which denotes the completion of a component that has been fully developed and is ready for integration into the next higher level of the system's organization. For example, the arrow ascending from the firmware stage represents the availability of the (actual) firmware. Accordingly, the integration process is ready to proceed to the next higher level (assuming the availability of the completed circuits as well) in which the firmware and circuitry will be combined in the machine stage to form the system's hardware configuration. The upward arrow from that stage indicates the hardware's readiness to be integrated with the (previously completed and tested) software. As is the case with design, there is no intent in the framework to imply a fixed dependency among the stages or phases. Parts of a project may move through their respective development processes faster than others, so that it may be reasonable for various parts to be in different phases/stages at a given time. This movement will be affected by the application and/or the characteristics of a particular methodology. Consequently, this diversity is not observed or delineated at the level of abstraction at which the framework functions.

C.3.2.2 Phases: Successful completion of each stage requires the completion of two or more constituent phases. (The individual phases are named in Figure C.1.) Although the activity associated with a phase is narrower than that of a stage, the relations among phases in a stage are conceptually similar to those that exist among the stages in the overall framework. Accordingly, each phase assumes an obligation to develop a conceptual, logical, or physical product that advances the state of the system and defines (completely) the work for the next phase. Similarly, when the system components are being integrated to produce the finished product, each phase assumes the responsibility for performing whatever integration is necessary (within its jurisdiction) to deliver its component or subproduct in final form as well.

To illustrate, the firmware design stage is isolated and redrawn with more emphasis on the individual phases (Figure C.2). (Deliberations undertaken during the machine design stage already would have established the advantages of firmware, thereby placing this issue beyond dispute in the firmware design stage.) The downward arrow from the microcode design phase denotes that phase's obligation to produce a complete set of

microcode requirements congruent with the firmware requirements delivered to it. In turn, these requirements serve as a basis for the microcode design to be turned out by the microprogramming phase. Exactly how the microcode requirements will be expressed and in what form they will be delivered depends on the particular methodology and its supporting facilities; the framework merely characterizes the activities.

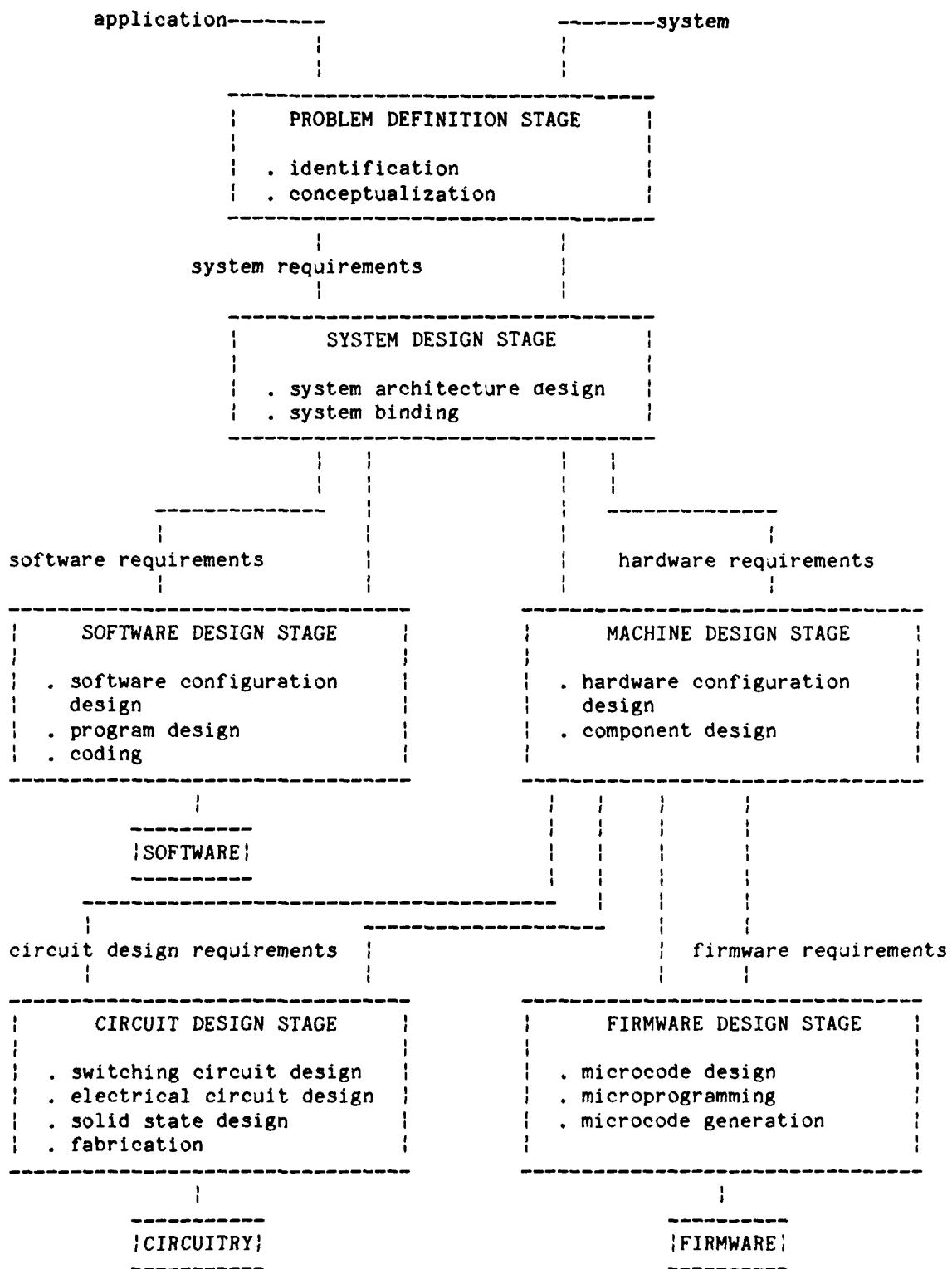
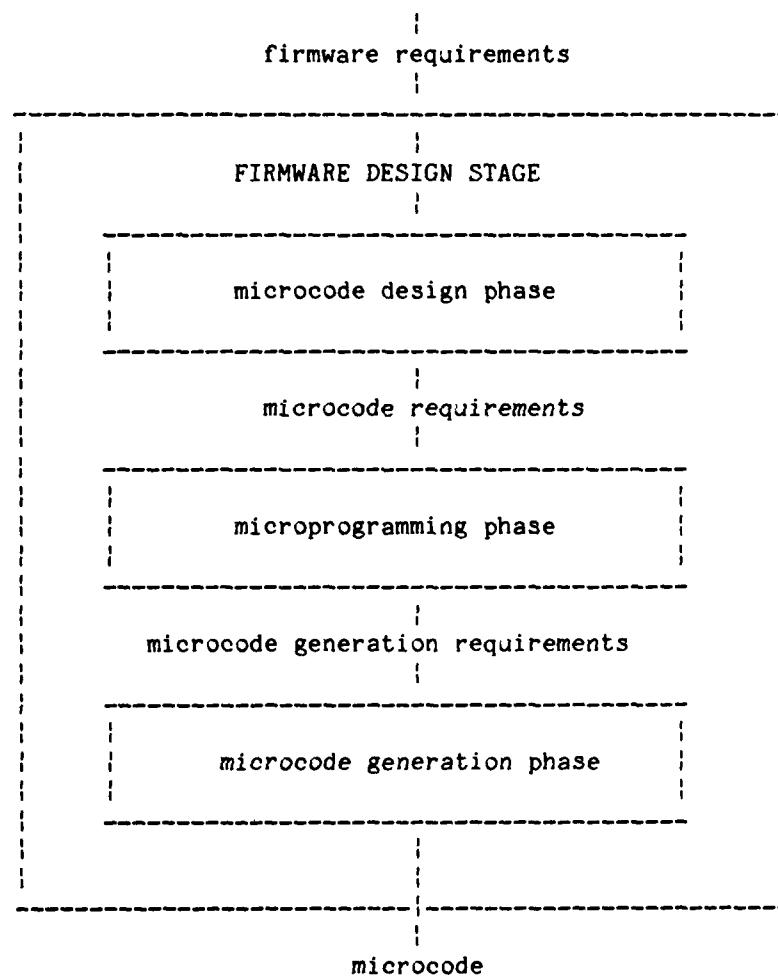


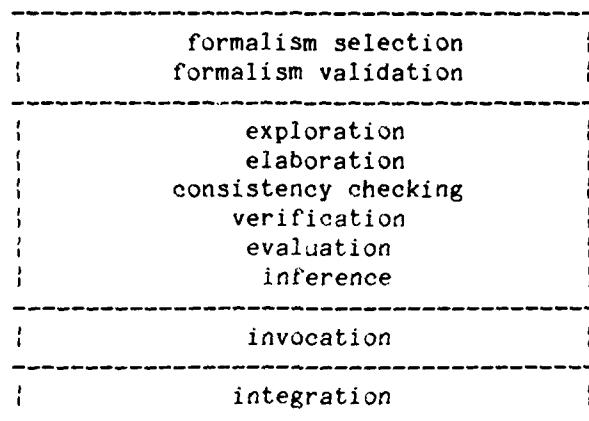
Figure C.1 TSD FRAMEWORK STRUCTURE

In the same way, the downward arrow from the microprogramming phase establishes the set of microcode implementation requirements as being the product delivered by that phase to motivate the production of the microcode itself by the third and final phase of the firmware design stage. The upward arrow from the microcode generation phase indicates that phase's responsibility to insure that its product (a set of micromodules) performs as an integrated whole in accordance with the implementation specifications on which they are based. The microprogramming phase, in turn, guarantees that those implementation specifications reflect precisely the intent of the microcode requirements.



Detailed Structure of the Firmware Design Stage
Figure C.2

C.3.2.3 steps: During the design process, each phase addresses a different design activity. However, the way the activity takes place is viewed by the framework as being conceptually consistent throughout all the phases. Accordingly, the framework subdivides each phase into a series of steps that are standardized across all phases. Some of the steps represent unit activities that reflect good practice and are fundamental to the design process in general. Others represent activities that support the realization of the framework's objectives in significant ways. The steps, named in Figure C.3, tend to cluster into four groupings, as the figure indicates.



Standardized Steps in Each Phase of the TSD Framework
Figure C.3

formalism selection

This step deals with the selection of an appropriate notational vehicle for expressing precisely the system component, requirements, or other entity that needs to be described by a particular phase. The suitability of a specific formalism will depend strongly on the problem domain for which it is to be used. Some will be especially simple and convenient to use for certain kinds of problems while being unnecessarily complicated and/or ambiguous for others. Often, the selection may not be made in isolation; rather, a particular formalism may be used because it is the one employed by a methodology chosen for other reasons.

formalism validation

Before a formalism can be applied, the adequacy of its expressive power must be determined. This is the purpose of the formalism validation step. In general, this activity involves a combination of theoretical and experimental assessments that examine the formalism's ease of use and potential for automation, as well as its ability to convey completely and precisely the information required from that phase. It is clear that the selection and validation steps are closely intertwined.

exploration

The exploration step, perhaps, is the most difficult activity to characterize. It is here that the germ of the design is born. Since this is a creative process, it has defied efforts at quantification and it continues to depend, in large measure, on talent, wisdom, and experience. Accordingly, it is irrelevant to consider candidate methodologies for quantifying this activity. However, the search continues for ways of lending some supporting structure to this step so that designers can focus their talents more effectively on the problems at hand.

elaboration

Design ideas produced in the exploration step are given form here. Depending on the phase being considered, elaboration may use vehicles ranging from mathematical formalisms to physical hardware in order to give expression to the ideas being developed.

consistency checking

This step encompasses activities centered around checking for incorrect uses of formalisms, detecting and reconciling contradictions, conflicts, and multiple viewpoints, and completing (previously) inadequate specifications.

verification

The purpose of this step is to demonstrate that a design embodies the functional properties called for in its requirements specification. For example, when applied in the program design phase, verification would seek to prove the correctness of the program specified there. Although such tasks continue to be inordinately difficult, it is important to note the step (and its intent) in the framework.

evaluation

Activities included in the evaluation step are aimed at determining whether a design meets a given set of constraints. These may stem from initial requirements, or they may be those imposed later in the system development cycle as the result of certain design decisions. Included are such constraints as system response time, throughput, fault tolerance, and cost. Accordingly, the evaluative approach will depend on the particular constraint being assessed. For example, investigation of system performance (in advance of the actual system's emergence) is likely to involve the use of simulation models, while cost analysis would call for an appropriate predictive model.

inference

In this step, the designers project beyond the immediate phase in which they are working to assess the potential impact of their design decision on other aspects of the system cycle and on the application environment itself. Since each phase interfaces closely with others, serious attention must be paid to the possible hardships imposed on the design activity in a subsequent phase by the choices made in a current one. Similarly, options selected to favor a particular design or performance attribute may exact an excessive (perhaps unacceptable) penalty on maintainability or enhanceability later in the life cycle.

invocation

The activity in this step centers around preparation of the phase's

product for formal release. In keeping with the philosophy intrinsically associated with methodological system design (Section C.1), such release must follow some type of "inspection" and "acceptance" procedures (their nature depending, of course, on the item being reviewed and the methodology under which the activity is organized). Once this release is secured, authorization to undertake the next phase is automatically implied. Thus, formal release of the specifications developed in one phase invokes subsequent phases.

integration

This final step encompasses the activities associated with the (conceptual or physical) assembly and testing of that portion of the total system designed in that phase. (This is in keeping with the assertion made earlier that responsibility for delivery of its product as an integrated, tested entity rests with each phase.) Because of the prominence of reliability as a system design objective, the TSD framework treats testing as an important activity whose proper exploitation requires the same expertise as does design. Consequently, integration is perceived as an activity to be localized at the phase level.

C.3.3 A Brief Walk Through the TSD Framework

Using the previous sections as background, we now can characterize the framework in more detail by examining each stage individually.

C.3.3.1 problem definition: Successful system design must start with a clear understanding of the problem being addressed. Although this need sounds self-evident, virtually every organization can cite instances of systems that do an effective job of solving problems other than those their users wanted solved. Consequently, as Figure C.3 indicates, the problem definition stage includes an explicit identification phase whose purpose it is to produce a written description of the problem that also specifies any basic constraints within which the problem must be solved. This report serves as the primary communication link between the users, who understand the application, and the analysts, designers, and developers, who are skilled in computer-based systems but need guidance with regard to the characteristics of the problem at hand. The pivotal importance of this document is underscored by the insistence that the concerned parties accept its description of the problem as being accurate, complete, and unambiguous.

identification phase

Although the identification phase is not a design activity, the written problem description developed here provides the basis for the entire system design. Hence, it is unique with respect to the breadth of issues it must address and the diversity of personnel who must agree on those issues. No single formal system could hope to fulfill the requirements imposed by this report. Consequently, the formalism selection and validation are not enforced as rigorously as they would be in more formal phases. English text, subject to structural constraints such as those imposed by report forms, outlines, and check lists, probably is the most appropriate vehicle for expressing the relevant assumptions,

constraints and demands to be satisfied by the proposed system.

The absence of a strict formalism does not lessen the importance of exploration, elaboration, consistency checking, and verification in this phase. User and builder must work together closely to establish the system's requirements, constraints, boundaries, and underlying assumptions. Trade-offs need to be anticipated and assessed to the greatest possible extent. In short, sufficient information has to be developed about the nature of the problem and the scope of the required solution to establish the beginnings of a project database that will support (and serve as the ultimate authority for) all subsequent project activities.

Completion of the identification phase involves acceptance of the identification report resulting from the detailed interactions between the system's developers and its ultimate users. Implicit in such acceptance is the authorization to begin (and eventually complete) the entire system development.

conceptualization phase

Using the rather informal system identification report as a basis, requirements now have to be expressed within a more formal conceptual model. The precise definition thus made possible will serve as the most direct source of direction for the system's design. Accordingly, the formalism selection and validation steps assume particular importance in this phase: Availability of a formalism that allows exact expression of the system's requirements increases the possibility that subsequent activities can be automated. For example, a set of precise syntactic rules (which such a formalism necessarily would include) allow automated consistency checking, documentation, and database updating. Consequently, there is a more objective foundation from which subsequent phases can draw. The effectiveness of a formalism is enhanced, incidentally, if particular care is taken to select a formalism that does not bias the resulting system definition toward a specific design alternative. Although it is complete and precise, the formal conceptual model of the proposed system still should not orient the system designer in any particular direction.

The value of the conceptual model becomes more apparent during the latter steps of the conceptualization phase. Here, the model is combined with the identification report to produce a set of system requirements. Fortified with this conceptual material, the report now includes a complete, unambiguous, testable set of requirements and constraints that the customer agrees will establish the formal basis for all later system development. Additional studies on cost effectiveness, scheduling, etc. (all started earlier) now may be expanded based on the more quantitative information developed from the conceptual model.

C.3.3.2 system design: This stage is central to the TSD framework because the logical design of the system is defined here. Hardware/software studies are conducted in sufficient detail to establish the manner in which hardware and software are to be used in the system's implementation. The primary motivation for this stage stems from the

specification of system requirements generated by the problem definition stage and qualified by performance constraints, time and financial considerations, and physical factors such as size, weight, and power consumption. In addition, the design process is guided by recognized principles and practices which promote the development of systems that resist obsolescence and are easy to maintain and enhance.

The activities of the system design stage can be characterized as a composite of the following concerns:

- a systematic examination of hardware/software tradeoffs;
- a systematic approach to system/environment interfacing;
- early definition of mechanisms for systematic error detection;
- identification of a disciplined approach to performance evaluation;
- explicit emphasis on maintainability and enhanceability;
- consideration of approaches that exploit computer-aided design.

The pivotal nature of this stage is reinforced by specifying its output requirements: The stage must produce all information needed for the design and/or procurement of the system's hardware and software. Moreover, it must develop sufficiently detailed system definitions for system integration, and it must identify specific procedural mechanisms for system maintenance and enhancement. Toward these ends, the system design stage is divided into two phases: the system architecture design phase and the system binding phase.

system architecture design phase

The system architecture design phase bears the responsibility of investigating system design alternatives and their potential impact on the choices for a feasible system configuration. Thus, consideration of hardware/software tradeoffs plays a crucial role here. Although this phase does not identify any specific hardware or software components, the hardware/software choices made here have a profound effect on shaping the class of eligible configurations.

Since the architecture specified by this phase will dictate the way in which the system is configured (but not the specific components that ultimately will be used), it is important to make sure that the formal description is complete. Consequently, the selection of the formalism must be able to represent the kinds of processing systems common to the application area. For example, if the project is in the C-cubed area, its architectural description is likely to require a formalism capable of representing networks of cooperating processors.

With the architectural specifications as a basis, the designer(s) can begin devising algorithms for performing the system processes defined in the conceptualization phase. The latter steps in this phase examine the resulting processing model to make sure that it does not violate the constraints established earlier and that it promotes maintainable and enhanceable implementations. Evaluation of the design and assessment of its implications for subsequent activities can take advantage of methodologies emphasizing the use of simulation.

system binding phase

Binding, in this context, refers to the process of defining hardware/software specifications that meet the processing requirements established in the architectural specifications. For some of the hardware, it may turn out that the system's requirements necessitate the use of a unique item, in which case the binding has already been done; there are no alternatives from which to select. This is unlikely to be true for all of the hardware; rather, the more common case is one in which several alternatives will be eligible candidates for a given component. It is the job of the binding phase to identify such candidates and select the most suitable one.

This selection process cannot be conducted for each component in isolation. A functional organization or fabrication technology that appears to be ideal for one part of the system may not be best for the system as a whole. Alternatively, it may violate an implementation constraint imposed by considerations elsewhere in the system. Placing the binding process at this point in the development cycle enforces the priority of system-wide considerations over local (component-level) issues.

Factors influencing the selection process include:

- product availability and manufacturer's marketing and fiscal position. This takes into account delivery time, usage history, and the manufacturer's ability to provide timely and effective support when there are problems.
- purchase cost for hardware and software. System-wide examination of cost factors provides opportunities to discover situations where more expensive alternatives for specific components allow the use of others that reduce the overall cost.
- operating cost. Included here are considerations such as power consumption, cooling, and other ongoing service costs.
- maintenance. Distinct from maintainability (i.e., the ease with which a system's or component's problems can be isolated), maintenance considers the activities involved in correcting problems. For example, the choice of a common vendor for functionally adjacent components may simplify maintenance procedures by reducing the

phenomenon known as "finger pointing."

-- enhanceability. Consistent with the anticipation that system requirements will grow as experience accumulates, it may be advantageous (where options exist) to favor hardware components that can grow easily. For example, a processor that is near the lower end of its architectural family may be preferable to a functionally acceptable alternative (which may even be less expensive) positioned at or near the top of its line.

When a required hardware/software component cannot be obtained off the shelf, the binding phase must produce specifications that are sufficiently detailed to enable contracts to be let for its design and development. Moreover, when custom hardware is to support software, the hardware/software interfaces must be sufficiently well-defined to allow concurrent and independent design of the software. Consequently, it is important to select specification formalisms that are adequate for expressing the varied functional and operational characteristics. Separate formalisms will be required for hardware and software descriptions.

Exploration is the key step in this phase. Although it would be ideal to specify custom hardware for all of the system's processes, reality dictates a philosophy that opts for customization only when there is no suitable alternative off the shelf. Consequently, many (if not most) of the hardware candidates are likely to include facilities that are irrelevant in the light of currently defined requirements but may be potentially useful later in the life cycle. Thus, the equipment with the fewest "extraneous" capabilities is not necessarily the most attractive choice.

The exploration step is complicated further by the need to recognize the strong hardware/software dependency: Choice of a particular hardware configuration constrains the design of the software that is to function with it. Conversely, if an available software product meets certain component requirements, its selection dictates (to an extent) the characteristics of the hardware with which it can be used.

C.3.3.3 software design: Using the software's functional and performance requirements developed during the system design stage, the software design stage prepares a working software system meeting those requirements. The resulting software constellation may take one of three basic forms:

- It may be a custom-designed system developed specifically for a project;
- It may be a prepackaged system acquired from an external source;
- It may be a combination of prepackaged and custom-designed components

When custom design is involved, the stage's activities are supported by an extensive accumulation of methodologies, facilities, and techniques that have been developed in response to a recognized need for handling increasingly complex software projects systematically.

Since the software design stage must produce a fully tested software package supported by appropriate documentation, there is particular emphasis here on making sure that the requirements and restrictions defined in the previous stage are reflected completely in the design and implementation of the resulting software products. Accordingly, integration and validation play important roles in the three phases that comprise this stage. The fact that there are three phases (with the actual coding being relegated to the final phase) underscores the importance of explicitly delaying the coding activities until the underlying design is completely defined and validated.

software configuration design phase

During this initial phase of the software design process, configurational requirements are transformed into a model of the final software system. A crucial aspect of this activity involves decomposition of the overall processing into interrelated functional components. This serves as a foundation for related activities within the phase to develop the following:

- a definition for the interfaces through which the components communicate with each other and with their environment;
- specification of appropriate performance requirements and special constraints related specifically to each of the functional components identified by the decomposition process;
- definition of the major data structures to be used in support of each functional module;
- identification of those components to be custom designed and those to be acquired from external sources.

As input to the program design phase, this material is the direct and immediate motivation for the programs' implementation. Consequently, the selected formalism must allow complete description of all of the design aspects. For example, if some or all of the modules must operate concurrently, the formalism must allow this concurrency to be expressed accurately and unambiguously.

Exploration and elaboration play particularly crucial roles in this phase. The decomposition process is by no means straightforward. Accordingly, great care must be taken to make sure that the resulting software design maintains a strong correspondence between conceptual activities and actual processes while providing the clearest, simplest interfaces among the functionally isolated modules. This could well require the development of several alternative software architectures

before a final selection is made. In addition to serving as the blueprint for subsequent implementation, this software design will guide the division of labor and definition of project schedules as well.

program design phase

The decomposition process continues in this phase. Using the architectural plan developed during the software design phase, each of the specified modules is developed in further detail, producing complete definitions of the appropriate algorithms and data structures. In addition, specifications for the connections among the modules are completely developed and documented. For many projects, it also may be appropriate for this phase to define evaluation procedures (and test data) for each of the software components.

Thus, once the work of this phase has been verified, there exists a complete set of software descriptions ready to serve as a basis for implementation. These are presented formally (using structured flowcharts, pseudocode, or some other suitable vehicle) so that there are no ambiguities with regard to the exact requirements; at this point, all of the software design has been done. In a real sense, the specifications produced by this phase are strongly analogous to detail drawings for a physical product's components.

the coding phase

The coding phase is responsible for the actual programming of the modules specified in the program design phase. In addition, it is responsible for the testing of those modules against their respective specifications. Formalism selection plays an interesting role in this phase: although the formalism must be taken from the domain of available programming languages, the selection itself may not occur during this phase. Rather, it may be imposed as a restriction during an earlier phase or even during an earlier stage. In fact, it is entirely possible for the programming language to be one of the constraints defined during the problem definition stage. Another interesting circumstance with regard to formalism selection in this phase is that it is possible to combine several formalisms without compromising consistency. For example, it may turn out that the implementation of a particular module requires the use of facilities accessible only through an assembler-level language while the bulk of the software can be coded satisfactorily in a higher level language.

The coding phase represents a logical terminus when considered within the overall TSD framework. That is, the phase's output consists of a set of validated, tested and documented software products not subject to further development as such. Rather, these components are ready for integration into the overall system: The initial aspect of this integration occurs within the program design phase where the individual modules are combined to determine the efficacy of their interconnections. This establishes and verifies the integrity of all of the custom-designed software. The resulting constellation, in turn, is combined with any off-the-shelf software to be integrated within the software configuration design phase. The result is the system's complete software. Final

evaluation of the total system then can take place in the system architecture design phase by combining software, firmware, and hardware to produce a prototype system which can be compared against its overall requirements and constraints.

C.3.3.4 machine design: As a result of the work performed during the binding phase of the system design stage the machine design stage starts with a clear definition of the system's hardware requirements. This includes an explicitly stated distinction between those components and/or subsystems that are to be procured off the shelf and those requiring custom design. Accordingly, the equipment in the former category is procured during this stage. In addition, a complete high level design is prepared for all custom hardware. This will serve as a basis for subsequent circuit design activity. If necessary, the stage also develops a set of firmware requirements for the hardware that has been purchased or designed.

The stage's work is divided into two phases: A hardware configuration design phase that produces component requirements, and a component design phase that produces circuit design requirements.

hardware configuration design phase

The purpose of this phase is to develop a formal model of the system at the hardware architecture level. The functional units comprising such a model typically are processors, memories, switching networks, input/output buses, and other intercommunication links. Consequently, the selected formalism is particularly important here: It must accommodate the full spectrum of these components. (For instance, a formalism designed to describe stand-alone computer systems is hardly suitable for a situation calling for distributed processing hardware.) If procured hardware is also part of the picture, the formalism must include capabilities for comparing properties and performance of existing machinery. A variety of hardware description languages are available as formalisms to support this effort. Once the formalism has been selected and validated, exploration is undertaken to identify eligible hardware to purchase. For custom-designed equipment, this step manifests itself as a modular approach to design in which expandable structures and simple interfaces are exploited to the greatest possible extent.

The overall hardware design thus produced must undergo comprehensive evaluation to make sure that its behavior and performance will meet the conditions and constraints imposed by the hardware requirements. Such evaluations combine tests on the actual hardware with those conducted on systems designed to simulate/emulate equipment not yet built or procured. For certain well-defined hardware configurations, it may be possible to evaluate performance analytically.

component design phase

Using the architectural requirements developed during the hardware configuration design phase, this phase procures the hardware that is off the shelf and redefines the components that are to be custom designed. Functional and performance requirements for these custom components now

are reflected in specifications at the circuit level. Here again, as in the configuration design phase, certain items will be identified as being available off the shelf while others will have to be custom designed. In this case, however, scrutiny occurs at the level of storage/shift registers, arithmetic/logical units, multiplexers, etc. rather than processors and storage structures. Thus, a custom-designed component may itself consist of prepackaged circuits combined with others developed explicitly for this system.

If some aspects of the system's operations are to be performed using firmware, this phase also carries those decisions forward by developing a set of firmware design requirements. These usually are expressed as an instruction set to be implemented via microprogramming on a host machine with a defined set of structural and behavioral properties.

The breadth of activities addressed during this phase may compel the use of several formalisms: Procurement of complete processors, input/output channels, etc. is supported by formalisms designed to allow precise hardware descriptions at that level. Such analytical vehicles are likely to be inappropriate when dealing with more primitive components at the register level. Consequently, a different formalism may be needed for those descriptions. Yet another formalism may be required for adequate expression of the architecture of the target machine (the machine to be represented by microprogramming) and its host (the equipment on which the firmware is to operate). Consequently, care must be taken to select formalisms that can be reconciled with each other when the diverse hardware components are integrated.

Exploration, elaboration, and consistency checking receive special emphasis in this phase because of the need to organize a collection of informal design requirements into a precise, cohesive set of specifications. For instance, this phase must make sure that the set of microinstructions defined for use in firmware production is completely compatible with the specified architecture of the host processor. Once the completed designs have been analyzed and evaluated (the latter activity usually involving tests on simulation systems), sufficient groundwork has been prepared to invoke the circuit design and firmware design stages.

C.3.3.5 the circuit design stage: This stage carries the circuit design process from requirements to actual fabrication. The work divides naturally into four phases, each of which is well supported by a variety of technological aids.

switching circuit design phase

The first major step toward fabrication of custom circuitry involves re-expression of the circuits' functions in terms of logic components such as operational amplifiers, flip-flops, and so on. Description of the circuits at this level enables designers to identify those logical components whose requirements can be fulfilled by available hardware. Formalisms used for such descriptions, including Boolean Algebra and switching-theoretic concepts, are well-defined and widely accepted.

Evaluation of the circuits at this level typically exploits technologies such as circuit simulation, breadboarding, and those derived from switching and graph theory. As a result, requirements for those circuits to be custom designed are defined and are ready for further work in the electrical circuit design phase.

electrical circuit design phase

Starting with logic circuit models augmented by performance requirements, this phase transforms these requirements into an equivalent set of solid state design requirements in which the circuits are described entirely in terms of conceptual electronic devices (e.g., transistors, resistors, etc.). Circuit details are expressed via standard electrical schematic diagrams enriched (as required) by formalisms that allow designers to model device characteristics and circuit layout geometry.

At this point in the custom circuits' developmental history, physical implementation assumes an important role in the design considerations. (For instance, use of discrete components over integrated circuits (or vice versa) will be determined during this phase.) This establishes a specific basis for defining the interconnections among the devices in each circuit.

the solid state design phase

This phase develops the finishing touches for the custom designed circuits, preparing them in a form suitable for fabrication: Device dimensions are fixed, detailed interconnections and chip geometry are worked out, and physical and performance characteristics of the final circuits are determined as accurately as possible. Using standardized design rules based on numerous successful precedents, all of the determinations are combined to form a precise specification of the geometry and layout for each circuit.

the fabrication phase

The final major activity in the circuit design process translates the geometric specifications into a physical reality. Using appropriate fabrication techniques (e.g., those employed in the preparation of an integrated circuit will differ basically from those applied to the synthesis of a circuit from discrete components), the actual hardware is produced and tested. Once the operational characteristics have been verified against their respective requirements, the finished circuits are available for integration with the rest of the hardware.

C.3.3.6 the firmware design stage: When deliberations conducted during the machine design stage determine that there is a need for firmware, appropriate requirements are generated and a distinct stage is included to translate those requirements into executable microcode supported by adequate documentation and analysis. Since the final output of this stage consists, basically, of programs, an analogy may be drawn between firmware development and general software development. In many respects, the concepts, techniques and tools seen to be helpful in the software area are applicable here as well. (This is reflected in the

similarity between the conceptual structure of this stage and that of the software design stage.) However, the direct relations between firmware and the low-level components of the machinery for which it is written often necessitate the use of code generation/optimization approaches not used in software development.

This stage is divided into three phases: the microcode design phase, the microprogramming phase, and the microcode generation phase.

the microcode design phase

Activities undertaken during this phase are roughly parallel to those associated with the software configuration design phase of the software design stage. Starting with a functional specification of the required firmware, this phase initiates the microprogram development process by preparing a program structure and decomposing it into functionally distinct modules. Performance requirements for each module are defined to a sufficient extent to allow appropriate algorithms to be identified during this phase.

To ensure a smooth and valid transition from microprogram design to actual microcode, the modules' specifications must be documented in a way that is consistent with the characteristics of the hardware on which the firmware will operate. Consequently, the selected description language is an important factor here. Use of a formally defined, machine-readable language is desirable since this enhances the prospects for automated verification. In any case, this phase must establish (to the greatest extent possible) that the microprograms produced here, when properly implemented, will meet the firmware's requirements.

the microprogramming phase

During this phase, functional modules designed during the previous phase are converted to individual statements in an appropriate microprogramming language. "Appropriate" in this context means that the language must be capable of conveying the full range of capabilities offered by the hardware on which the final microcode is to be executed. Moreover, the language must be supported by an available translating vehicle (usually software) that will produce functionally equivalent, executable microcode. In this phase, the firmware development process has progressed to a point where all design decisions have been made, and attention can focus on implementation issues such as selection of storage structures and subroutine linkages.

the microcode generation phase

In a conceptual sense, this phase "fabricates" the actual microcode by processing the microprogramming statements through a suitable translator. Although the activity is essentially automated, there may be instances whereby a particular performance constraint may be violated because of inefficiencies present in the microcode generator. Such shortcomings often are remedied by manual adjustment of the microinstruction sequences. In effect, this subjects the automatically produced code to a "fine tuning" process.

As is true with hardware and software design, the products developed here are in their final states (as components) in that they are not subject to further dissection. Rather, they are ready to enter the integration process in which firmware modules will be combined to form the overall firmware configuration which, in turn, will be installed on a host processor to provide a suitable target machine for the system's software.

C.4 Existing Resources for Methodological System Development

Response to acknowledged problems in system design/development has been characterized (Section C.2.3) by changes in designers' and managers' perception of computer applications and their realization. This has prompted the development of methodologies and tools aimed at supporting a disciplined approach to the system development process.

In a sense, the appearance of these software resources is not a revolutionary development. Rather, it can be viewed as a stage in a continuing process that started with the first assembler-level programming language. The "revolution" occurred at that point because the basic relationship between the programmer and the machine suddenly was changed by the introduction of a "third party" - a software product designed to facilitate coding. This third participant has been an integral part of the picture ever since, its scope continually broadening to include aid for more and more of the activities performed throughout the system cycle: in addition to providing growing support for the programming process (realized via increasingly powerful and convenient programming languages), tools and methodologies have been introduced to aid in system specification, design, documentation, and testing as well. In fact, the level of support has reached a point [HOWD81] where it now is appropriate to view a collection of such aids conceptually as a software development environment. As a result, today's designers can call on a variety of resources ranging from simple checklists to comprehensive software packages equipped with extensive automated features to support documentation and checkout(*).

Unfortunately, there is no single methodology that has been found to be the "best" vehicle for orderly system development, nor is it likely that one will emerge. The existence of numerous "competing" approaches, all of them orderly, attests to the unavoidable fact that the effectiveness of a particular methodology depends strongly on the kind of system to which it is applied. For example, a software facility useful in documenting batch-oriented data processing systems may prove inadequate for expressing some of the realtime properties that are basic to a C-cubed system. Moreover, a point made in Section C.3 is worth reemphasizing here: support over the entire range of a project's activities requires a combination of methodologies, each of which supports only part of those activities. Consequently, selection of appropriate compatible

* The National Bureau of Standards [HOU80] has undertaken the task of serving as a clearinghouse for documentation on software methodologies, tools, and techniques.

methodologies can exert considerable influence on the system's progress. The TSD framework can be of considerable help in systematizing this selection process; its use for this purpose is examined in Section C.4.1. Section C.4.2 then mirrors available tools and methodologies against various parts of the framework to characterize the nature and extent of existing software engineering support. Finally, Section C.4.3 looks at general (hardware/software) facilities that are currently available for support of system development.

C.4.1 Relation between the Framework and Applicable Methodologies

The TSD framework serves as a convenient guide upon which various methodologies can be superimposed and compared. As indicated earlier, the TSD framework's organization provides definitions for a complete hierarchy of stages, phases, and steps. Thus, an effective first step in applying the framework is to juxtapose these definitions against a particular application. This can identify phases (or, perhaps, entire stages) whose activities are unnecessary for that project.

For instance, if the project calls for the production of an easily replicated turnkey system consisting of one or more processors equipped with predefined small-scale data processing programs, the machine design stage can be eliminated. Unless the applications' requirements are particularly severe or exotic, there is no reason for an organization to assume the considerable additional burdens imposed by the preparation, checkout, and subsequent maintenance of a new machine. Instead, processing hardware can be selected from existing choices during the system design stage. Once the decision has been made to use off-the-shelf equipment, the machine design stage drops out, and the circuit design and firmware design stages are eliminated automatically. With certain parts of the framework having been earmarked as superfluous for a given project, the remaining stages and phases can be analyzed to establish their respective roles in the project. To illustrate, let us pursue the data processing project mentioned before. Since the hardware requirements are to be met (in that example) by predesigned equipment, the system design stage is less comprehensive than it would be for a situation in which the use of such hardware is uncertain or infeasible. Specifically, the system architecture phase can be limited to a determination of the number of (identical) processors over which the system's operations will be distributed, along with the nature of the distribution. The binding phase, then, will match the desired hardware characteristics against those of eligible processors, and it will generate appropriate hardware and software requirements.

These considerations, applied to each relevant stage and phase, enable design and management personnel to identify appropriate formalisms for describing the outcomes of the respective activities. In many organizations, the scope of system designs is sufficiently restricted to allow certain formalisms to be defined as standards for various phases. For instance, it may be possible to standardize on a particular form of pseudocode for the program design phases (regardless of the project) and/or on a particular program language for the coding phase. This simplifies matters even further since it obviates the need to include formalism selection and validation steps in those phases to which the

standard formalisms apply.

Next we need to consider evaluation of the descriptions to be voiced using the selected formalisms. Employing the (reduced) framework again, each phase can be associated (if appropriate) with approaches and techniques deemed suitable for the nature and extent of the activities undertaken in that phase for that project. For example, evaluation in the conceptualization phase may involve a user review; evaluation in the coding phase may take advantage of automated debugging software, while evaluation in the other phases of the software design stage may be ignored altogether as being irrelevant.

In this way, developmental aids, each designed to support particular activities, can be brought together and considered within the comprehensive structure of the TSD framework. When the selections are made, the result is a cohesive methodology, tailored for the project and designed wherever possible to exploit available tools and techniques to facilitate the system's implementation and management.

C.4.2 Software Engineering Support

It would be misleading to create the impression that the system designer, eager to put together an effective methodology for his or her project, simply selects from a wide variety of existing, proven aids and techniques. True, there are numerous software engineering resources, and some of these have demonstrated their utility. (The National Bureau of Standards estimates that there are over 400 commercially available software development tools [HOWD81]). However, many of these aids are effective only for certain families of applications. Moreover, there are entire areas within the framework that continue to be resistant to analysis and quantification. For these activities, methodological aids remain largely manual, and many exhibit essentially a qualitative character. This does not necessarily detract from their importance. The major point is that such aids can be effective if they succeed in eliminating (or drastically reducing) the "ad hoc" aspect of the associated activity.

C.4.2.1 aids for problem definition: The "system first" approach to project design, pioneered by the Air Force [CLAR79a], is a relatively new idea. It is not surprising, then, that current software engineering support for the problem definition stage is base predominantly on the premise that the hardware is already prescribed and the "problem" being defined will be solved by software. Consequently, the ability to apply a particular aid to a given project cannot be assured.

During the identification phase, it is desirable to find ways of removing ambiguity from requirements definitions and providing opportunities for disciplined review and revision. Since the major communication vehicles are documents containing informal descriptions, aids seek to promote more controlled documentation in two related but distinct ways. One objective is to provide a computer-based operating environment for the orderly creation, updating, and management of documents. Such facilities, designed for use with any document collection, include text editors, formatters, report generators, and

library maintenance software. These are available in abundance and are usually tailored to particular computing systems. (Of course, general resources for documentation support will find similar uses in every stage and phase.)

A second (complementary) approach to support during the identification phase seeks to introduce structure into the documents themselves through such organizing features as check lists, controlled vocabularies, standardized forms, and other, similar features designed to make the writer more aware of what he or she says and how he or she says is [HENI79, RAMA78, TAGG79].

Facilities applicable to the conceptualization phase tend to be specification languages with sufficient formalism to allow automated checking for syntax and completeness. Prominent examples of such resources are given in [LISK79, ROSS77b, TEIC77].

C.4.2.2 aids in system design: The all-important hardware/software tradeoffs, considered during this stage, involve activities in which experience, judgment, serendipity, and other intangible factors play a significant role. Consequently, it would be unduly optimistic to expect anything substantial by way of automated support for this inherently complex process. However, powerful auxiliary aids exist to help facilitate and organize the description of hardware/software requirements produced by this stage.

During the system architecture phase, designers decide how to distribute the system's processes between hardware and software. These decisions must be expressed as precise descriptions of function and performance requirements for each process so that they may drive hardware selection and software design. Numerous software-based facilities have been designed to help turn the architects' decisions into effective blueprints. On the hardware side, there are description languages [SHIV79] for defining individual processors as well as networks of interconnected machines. These languages are equipped with processing software for automatic consistency checking. In addition, the general documentation aids mentioned before often include capabilities for representing, updating, retrieving and displaying design material in graphical form [ROSS77b]. Software description languages [LISK79, BELL77, TEIC77, CAMP79] also include their own processors for consistency checking.

The binding phase, during which requirements for processors and interconnections must be matched to suitable candidates, is not well supported at present. Processing requirements defined during the architecture phase necessarily are specific to the needs of the system as perceived at that point in the project's history. Consequently, a search among existing computers is likely to identify candidates whose features, though adequate for the job at hand, also include those that appear to be "irrelevant" when viewed against the hardware requirements. Yet, their potential usefulness argues against immediate rejection of such candidates. This complicates the binding process so that proven methodologies are not available [TIMM73]. However, the use of grammatically consistent hardware description languages in the

architecture phase places the comparison of hardware candidates on a more organized basis.

C.4.2.3 aids in software design: This area of the system cycle has received tremendous attention. In fact, what is now beginning to be recognized as a system crisis was perceived originally as a software crisis [ROMA81]. As a result, there is an extensive repertoire of methodologies dealing with the programming/coding process and, more recently, with the broader scope of software design. Unlike the other stages in the framework, software design includes aids for project management as well as those resources with a technical focus.

The first of this stage's three phases (i.e., software configuration design) can avail itself of a variety of methodologies, all of which are intended to organize and facilitate the task of producing a coherent structural model of a software system and its components [ROSS77a, ROSS77b, MYER73, IBM74, GANE79, LISK77b, TEIC77, HAMI76, BELL77]. In addition, tools are available for setting up disciplined, systematic design reviews [YOUR75].

The program design phase, charged with the transformation of the program design requirements into a set of implementation requirements, also may call on an extensive array of technical supports. There are guidelines and techniques [PARN72, WIRT71, BERG81] for facilitating the process of decomposing a set of software requirements into functionally distinct modules. Description of the results may take one of several well-defined graphical forms [NASS73, JACK75], or the designers may find it more advantageous to use a more formal description language [TEIC77, AMBL77, PAGA81] with opportunities for automated consistency checking.

The final phase, during which the actual code is produced, has been the focus of the structured programming revolution, a movement that has produced profound changes in programming techniques [WIRT73, KERN74, DAHL74, GILL82], programming project management [BAKE72, BRO075, ROCH75] and programming languages themselves [DAHL66, JENS75, DOD79, POLL82]. In addition, considerable understanding has been gained with regard to systematic program testing, so that test design methodologies [HUAN78, GILL82] can be used to expedite and improve testing (both at the module and system levels) during maintenance as well as development. Although complete formal program verification is not a reality, the insights developed from research in the area [ROBI77, WEGB76] have contributed substantially to the improvement of program description and implementation methodologies.

C.4.2.4 aids in machine design: Although a wealth of experience has been accumulated in the design of custom digital hardware, it is only recently [SHIV79] that concerted efforts have been mounted to provide formal specification methods and a cohesive support facility for hardware development. Now, with added impetus from the American National Standards Institute, definitions for a universally acceptable formal symbology for hardware description are well underway, and the availability of design support facilities meeting TSD requirements are nearing realization.

Work performed during the hardware configuration design phase can benefit from the same array of hardware description languages [SHIV79] applied to the system binding phase. These languages include facilities for expressing hardware requirements at the component level and below. Moreover, their software support provides automated consistency checking for the resulting component specifications.

Support for performance evaluation remains elusive at present. A number of analytical and simulation approaches have been suggested [TO74, COX79], but adequate attention to this important activity ultimately may be found in the availability of versatile hardware-based emulation facilities.

As mentioned earlier, currently available hardware description languages also serve as satisfactory vehicles for the detailed specification of components to be custom designed. This is not the case with regard to any microprogrammed control structures defined during this phase. Descriptions of such components necessarily must be expressed as a set of target instructions to be implemented (in firmware) on a host machine. The activities leading to such a definition are subject to the same intangible forces that operate in the system architecture phase. Accordingly, absence of supporting methodologies is not surprising.

Solid support for the evaluation of component designs also is unavailable at present. As in the case of higher level hardware designs, a satisfactory answer eventually may be found in emulation.

C.4.2.5 aids in circuit design: As is true with software design, these activities have received an enormous amount of attention. Unlike software design, however, the payoff has already arrived and is well established [MEAD80]. Circuit design and fabrication are so well supported that any discussion regarding the application of a methodology would be anachronistic. Consequently, no purpose would be served in pressing the issue further in this Appendix.

C.4.2.6 aids in firmware design: Current support for firmware design is conceptually similar to that available for software design. (This is understandable since the respective activities in the two stages are strongly analogous to each other.) Accordingly, the designer can call on the general documentation/library management methodologies and accompanying software used throughout the system cycle. In addition, there are language aids designed specifically to facilitate the process of transforming a set of firmware requirements, expressed in one language, to a functionally equivalent set of coded microinstructions in the host machine's (low level) language.

The stage's initial phase, charged with the development of an integrated microprogram design, can avail itself of a broad range of methodologies (such as those cited in Section C.4.2.3) for help in decomposing the firmware specifications into a coordinated set of functionally distinct modules. Of special interest here is the availability of high level microprogramming languages (SMITE [SMIT77] being the most completely defined) as convenient vehicles for expressing microprogram designs. In some situations (i.e., for certain host machine

architectures) it may be possible to submit the high level language description directly to a compiler, in which case the production of executable microcode is completely automatic. This eliminates the need for the interim transformation otherwise handled by a separate microprogramming phase.

Since a system's firmware requirements are likely to include stringent performance constraints (e.g., maximum allowable execution times for target instructions), automatic production of executable microinstructions often is followed by an effort to optimize the microcode. This process may or may not be manual, depending on the availability of optimizing software for the particular host machine [TOKO78].

C.4.3 Facility Support

The document handlers, text formatters, assemblers, compilers, consistency checkers, debugging packages, data base managers, performance monitors, and other computer-based software engineering aids outlined in the previous section exist in the form of programs implemented on one or more types of general purpose computer systems. With a "hardware first" philosophy, this has meant that, once the computer type was selected, the choice of software-based aids would be determined by the range of products available for that machine. Moreover, the extent to which available software engineering support is exploited on a given hardware system often is affected by short-range economic pressures. These considerations tend to limit hardware procurement to equipment that is adequate for system testing/modification under hot bench conditions, but is insufficiently configured to accommodate the full complement of additional resources that offer helpful technical and managerial support. For example, a configuration explicitly equipped to exploit systematic design/development aids is likely to include peripheral devices (and ports to accommodate them) for programmers' work stations, massive online secondary storage for such items as source code libraries, system documentation and project histories, and additional main storage for support programs. These resources would be considered superfluous (and intolerable) in a weapon's embedded computer system or in an airborne data acquisition system.

Consequently, most of today's facility support can be characterized as being project-specific: For a given project, the facility consists of that project's hardware (or a realistic surrogate) implanted with a (limited or extensive) software development environment. (Since the hardware is already defined, consideration of an analogous hardware development environment is, by definition, irrelevant in this context.)

Proliferation of software development methodologies and tools, coupled with a growing recognition of their effectiveness, has prompted growing interest in hardware/software complexes designed explicitly to be software development facilities. Such a facility would be equipped to accommodate a number of groups working concurrently (and independently) on different project for different hardware using different software development environments.

Perhaps the most prominent instance of this type of centralized facility is the Naval Air Development Center's Facility for Automated Software Production [FASP]. Although designed with a "hardware first" orientation, the facility offers a degree of flexibility by accepting software intended for any of several standard military computer systems. The central computing constellation, though architecturally different from the target machines, provides cross-assemblers and cross-compilers that enable users to write code in standard high level programming languages and have that code translated automatically to instructions that will execute on their respective hot bench configurations. A range of software-based development/management aids also is available so that the individual project can choose a methodology that provides the most appropriate support. Typical usage involves remote access to the FASP (through the use of secure data communication lines) in conjunction with the hot bench system on site.

The Air Force is looking to carry this concept further [CLAR79b, CLAR79c] by expanding the resources of such a facility to include microprogramming capabilities for emulating a potentially arbitrary range of hardware architectures. Properties of this type of facility, and its relation to the TSD philosophy, will be examined in the next section.

C.5 Future Trends in System Design Methodology

Although it is difficult to chart an orderly future, especially in a dynamically changing field such as computing, sufficient momentum has built up in the system design area to provide a reasonable basis for extrapolation. The factors that brought on the system crisis (Section C.1) will not go away: It is clear that requirements for computer-based systems continue to grow in complexity, thereby placing increasing pressure on design organizations to find and use more cost-effective ways of producing reliable systems that can be maintained and enhanced without cataclysm. At the same time, advances in microelectronics also contribute to designers' mounting technological difficulties by increasing the number of serious design alternatives for a given system.

These combined forces help give impetus for the continuing movement of the system development process away from its ad hoc roots. In this connection, the already sizable array of computer-based design tools is certain to be augmented by a continuing stream of new text editors, report generators, automated librarians, and so on. More significantly, an increasing number of these new products will be designed to provide support for the first three stages of the system cycle - areas where such support currently is sparse.

This growing emphasis on the conceptual aspects of the system design process can be viewed as a clear signal that the next major stage in the evolution of system design will be a shift from a "hardware first" philosophy to a "system first" orientation in which the hardware/software duality is explicitly recognized, and ample opportunities are provided for full exploitation of systematic methodologies. Although the need for such a shift is compelling, the actual transformation will be painful. In order for such a change to be realized, designers must be provided with

computer-based facilities that differ from their predecessors in three important respects:

- Instead of concentrating primarily on software development, the next generation of facilities will have to provide total system design environments in which integrated sets of services and tools support a project's activities over the entire system cycle - from conception onwards.
- Total system design facilities must respond to the fact that the effectiveness of a particular methodology will depend (sometimes strongly) on the application for which its use is being considered. Consequently, such facilities cannot limit themselves to a single collection of tools that form an "official" methodology. Instead, there must be a range of computer-based tools, installed within a powerful logical framework that enables designers to select those that are appropriate to their needs and integrate them into an effective methodology. Thus, the facility provides the means whereby each project can construct (and operate in) its own system design environment.
- There is every likelihood that newly emerging tools will offer improved vehicles for supporting various activities in the system cycle. For example, research in specification languages promises to motivate important progress in automated documentation and verification mechanisms. Consequently, a Total System Design facility will have to be inherently dynamic - capable of accepting new tools and making them available for integration into more effective methodologies.

An important technology that will be exploited extensively in the forthcoming "system first" support facilities is that of emulation - the ability (via microprogrammed firmware) to represent the internal functional structure of one machine (the target machine) on another machine (the host) that is architecturally different. The role of emulation is seen to be twofold: First, it enables a particular computer system to accept a broader range of (existing) software products by providing "machines" that are "identical" to those for which the software was implemented. This offers designers enhanced opportunities for the preparation of effective system design environments.

The second role, more significant in terms of its impact on the system design process, places hardware selection on an equal footing with that of software. Functional characteristics of alternative architectures can be compared without the time and expense involved in obtaining the actual hardware. This obviates the need to "settle for" a particular configuration so that it can be procured and delivered in time for software and performance tests. Once the architecture is selected, the emulated system continues to serve as a vehicle for software development

and testing in advance of the actual hardware's arrival and installation. In fact, the operating characteristics of an emulated system can be so close to its actual counterpart that it often is possible to use the emulated system for obtaining valuable data on system performance as well as operational integrity. Of course, the criteria for system operability still must be tied to hot bench results; however, the important point is that the availability of an emulated system allows the hardware and software procurement cycles to proceed in parallel, thereby opening the area of system architecture to exploitation.

DOD has been early in recognizing the need to move away from the increasingly burdensome constraints imposed by a facility with fixed architecture(s). Accordingly, it has pioneered the development of facilities in which emulation is included among the resources. The System Architecture Evaluation Facility (SAEF), located at RADC, is a prominent example. SAEF is designed to provide an experimental laboratory for research into the advanced hardware configurations necessary to support the complex information processing needs of military command, control, and communication systems. It allows overall system designs and alternatives, both hardware and software, to be evaluated quickly and conveniently, thereby minimizing actual development and life-cycle costs for new systems. Construction of actual new hardware components can be delayed or avoided by providing means for emulating such components on a microprogramming computer (a Nanodata QM-1). Thus, programs may be written for a proposed machine design at the machine language level and then executed by the microprogrammed machine emulation.

The QM-1 is operable in both a standalone mode (under which conditions it supports a full complement of peripheral equipment) and in a timesharing mode connected to a DECSYSTEM-20 computer. The latter is equipped with many supporting tools that provide the necessary control and reporting capabilities for the installation and enable users to interact with the system easily and conveniently.

The architecture to be emulated is defined in a high level language named SMITE (Software Machine Implementation Tool for Emulation). These descriptions (specified at the register transfer level) are compiled into microcode to run on the QM-1. Still to be produced is a versatile language processing system that would automatically compile object code for a target machine based on a SMITE description of that machine.

The SAEF now is perceived as the basis for a more comprehensive resource that will support a computer-oriented project by enabling its workers to do their jobs within productive environments consistent with the TSD framework. Chapter 4 of this Report describes the master plan for such a facility (the first of its kind) in detail. In addition to providing a wide range of existing tools, the TSD facility would be designed to accept new tools for general use as well as specialized tools needed for particular projects. Integration of a given set of tools to form a suitable environment for a certain system activity (management of documentation versions, for example) will be aided by a command language through which the user avails himself of the facility's resources. Commands submitted in the same language would enable another project member to create a different environment effective for his or her

particular activity (e.g., preparation of an emulator for an architecture to be tested). Consistency and control is to be maintained by storing all information about a project in a data repository managed by effective database software.

Initially, the TSD facility is to support the problem definition, system design, and software design stages of the TSD framework, with expansion to the other stages to follow in due course. Although it may be possible to employ this facility to support an actual DoD project, it is intended primarily to serve as a prototype - a test bed for investigating the organizational and behavioral properties of such facilities. Information gathered by such inquiries is expected to provide definitive guidelines for the preparation of subsequent, more powerful TSD facilities for actual project support. There is reason to be optimistic that the emergence of increasingly systematic design methodologies, together with effective facilities to support them, will contribute significantly to the attenuation of the system crisis.

Bibliography

- [AMBL77] Ambler, A. L., et al, "Gypsy: A Language for Specification and Implementation of Verifiable Programs," ACM SIGPLAN Notices, Vol. 12 No. 3 (1977).
- [BAKE72] Baker, F. T., "Chief Programmer Team Management of Production Programming," IBM Systems Journal, Vol. 11 No. 1 (1972) pp 56-73.
- [BELL77] Bell, T. E., Bixler, D. C., and Dyer, M. E., "An extendable Approach to Computer-aided Software Requirements Engineering," IEEE Trans. on Software Engineering, SE-3, No. 1 (1977) pp 49-60.
- [BERG81] Bergland, G. D., "A Guided Tour of Program Design Methodologies," Computer, Vol. 14 No. 10 (1981) pp 13-37.
- [BOEH73] Boehm, B. W., "Software and its Impact: A Quantitative Assessment," Datamation, Vol. 19, No. 5, May 1973, pp 48-59.
- [BRAN81] Branstad, M. A., and Adrion, W. R. (eds.), "NBS Workshop Report on Programming Environments," Software Engineering Notes, Vol. 6 No. 4 (1981) pp 1-51.
- [BROO75] Brooks, F. P., Jr., The Mythical Man-Month, Addison-Wesley (Reading, MA), 1975.
- [CAMP79] Campbell, R. H. and Kolstad, R. B., "Path Expressions in Pascal," Proc. 4th International Conference on Software Engineering, (1979), pp 212-219.
- [CLAR79a] Clark, N. B., The Total System Design Methodology, White Paper, Rome Air Development Center, 1979.
- [CLAR79b] Clark, N. B., Common Software Support Environment, White Paper, Rome Air Development Center, 1979.
- [CLAR79c] Clark, N. B., and Troutman, M. A., The System Architectural Evaluation Facility and Emulation Facility at Rome Air Development Center, White Paper, Rome Air Development Center, 1979.
- [COX79] Cox, L. A., Jr., Performance Prediction from a Computer Hardware Description, Report No. NPS52-79-001, Naval Postgraduate School (Monterey, CA), 1980.
- [DAHL66] Dahl, O. J., and Nygaard, K., "SIMULA - An ALGOL-based Simulation Language," Communications of the Association for Computing Machinery, Vol. 9, No. 9, September 1966, pp 671-678.

[DAHL74] Dahl, O. J., Dijkstra, E. W., and Hoare, C. A. R., Structured Programming, Academic Press (New York, NY), 1974.

[DOD79] Department of Defense High Order Languages Working Group, Preliminary Ada Reference Manual, ACM SIGPLAN Notices, Vol. 14 No. 6, Part A (1979).

[FASP] The Facility for Automated Software Production, Advanced Software Technology Division, Naval Air Development Center, Warminster, PA.

[GANE79] Gane, C. and Sarson, T., Structured Systems Analysis: Tools and Techniques, Prentice-Hall (Englewood Cliffs, NJ), 1979.

[GILL82] Gillett, W. D., and Pollack, S. V., Introduction to Engineered Software, Holt Rinehart and Winston (New York, NY), 1982.

[HENI79] Heninger, K. L., "Specifying Software Requirements for Complex Systems: New Techniques and Their Applications," Proc. Conf. on Specification of Reliable Software, April 1979.

[HOWD81] Howden, W. (ed.), "Contemporary Software Development Environments," Software Engineering Notes, Vol. 6 No. 4 (1981), pp 6-15.

[HUAN78] Huang, J. C., "An Approach to Program Testing," in Software Methodology (C. V. Ramamoorthy and R. T. Yeh, eds.), IEEE Catalog No. EHO 142-0 (1978) pp 318-333.

[HOUG80] Houghton, R. C., and Oakley, K. A., NBS Software Tools Database, National Bureau of Standards Report NBSIR 80-2159, (Washington, D. C.), 1980.

[IBM74] HIPO - A Design and Documentation Technique, GX20-1851, International Business Machines Corp. (White Plains, NY) 1974.

[JACK75] Jackson, M. A., Principles of Program Design, Academic Press (New York, NY), 1975.

[JENS75] Jensen, K., and Wirth, N., PASCAL User's Manual and Report, Springer Verlag (Berlin), 1975.

[KERN74] Kernighan, B. W., and Plauger, P. J., The Elements of Programming Style, McGraw-Hill (New York, NY), 1974.

[KERN81] Kernighan, B. W., and Mashey, J. R., "The UNIX Programming Environment," Computer, Vol. 14, No. 4, April 1981, pp 12-24.

[LISK77] Liskov, B., and Zilles, S., "An Introduction to Formal Specifications of Data Abstractions," in Current Trends in

Programming Methodology, Vol. 1, R. Yeh (editor), Prentice-Hall (Englewood Cliffs, NJ) 1977.

[LISK79] Liskov, B. H. and Berzins, V., "An Appraisal of Program Specifications," Research Directions in Program Technology, MIT Press (Cambridge, MA), 1979, pp 276-301.

[MEAD80] Mead, C., and Conway, L., Introduction to VLSI Systems, Addison-Wesley, (Reading, MA), 1980.

[MYER73] Myers, G. J., "Composite Design: The Design of Modular Programs," Technical Report TR002406, IBM (Poughkeepsie, NY), 1973.

[NASS73] Nassi, I., and Schneiderman, B., "Flowcharting Techniques for Structured Programming," ACM SIGPLAN Notices, Vol. 8 No. 8 (1973) pp 12-26.

[PAGA81] Pagan, F. G., Formal Specification of Programming Languages, Prentice-Hall (Englewood Cliffs, NJ), 1981.

[PARN72] Parnas, D. L., "On the Criteria to be Used in Decomposing Systems into Modules," Communications of the Association for Computing Machinery, Vol. 15 No. 5 (1972) pp 1053-1058.

[POLL82] Pollack, S. V., Structured FORTRAN 77 Programming, Boyd and Fraser (San Francisco, CA), 1982.

[RAMA78] Ramamoorthy, C. V. and So, H. H., "Software Requirements and Specifications: Status and Perspectives," in Software Methodology, IEEE Catalog No. EHO 142-0 (1978), pp 43-165.

[ROBI77] Robinson, L., and Levit, K. N., "Proof Techniques for Hierarchically Structured Programs," Communications of the Association for Computing Machinery, Vol. 20 No. 4 (1977), pp 271-283.

[ROCH75] Rochkind, M. J., "The Source Code Control System," IEEE Transactions on Software Engineering, Vol. SE-1, 1975.

[ROMA81] Roman, G-C et al, TSD Methodology Assessment, Interim Report, Contract F30602-80-C-0284, Washington University (St. Louis), 1981.

[ROSS77a] Ross, D. T. and Schuman, D. E., "Structural Analysis for Requirements Definition," IEEE Trans. on Software Engineering, SE-3 No. 1 (1977), pp 6-15.

[ROSS77b] Ross, D. T., "Structured Analysis (SA): A Language for Communicating Ideas," IEEE Trans. on Software Engineering, SE-3, January 1977.

[SHIV79] Shiva, S. G., "Computer Hardware Description Languages - A Tutorial," Proceedings of the Institute of Electronic and

Electrical Engineers, Vol. 67 No. 12 (1979), pp 1605-1615.

[SMIT77] SMITE Training Manual, Report 30417-6002-RU-CO, 1977.

[TAGG79] Taggart, W. M. and Tharp, M. O., "A Survey of Information Requirements Analysis Techniques," Computing Surveys, Vol. 9 No. 4 (1977).

[TEIC77] Teichrow, D. and Hershey, E. A., "PSL/PSA: A Computer Aided Technique for Structured Documentation and Analysis of Information Processing Systems," IEEE Trans. on Software Engineering, SE-3, January 1977.

[TIMM73] Timmreck, E. M., "Computer Selection Methodology," Computing Surveys, Vol. 5 No. 4 (1973) pp 199-222.

[TO74] To, K., and Tulloss, R. E., "Automatic Test Systems," IEEE Spectrum, September 1974, pp 44-52.

[TOKO78] Tokoro, M., Takizuka, T., Tamura, E., and Yamaura, I., "A Technique of Global Optimization of Microprograms," Proceedings of the 11th Annual Workshop on Microprogramming, (ACM), 1978, pp 41-50.

[WEGB76] Wegbreit, B., "Verifying Program Performance," Journal of the Association for Computing Machinery, Vol. 23 No. 4 (1976), pp 691-699.

[WEIN70] Weinberg, G., The Psychology of Computer Programming, Van Nostrand Rheinhold, (New York, NY), 1970.

[WIRT71] Wirth, N., "Program Development by Stepwise Refinement," Communications of the Association for Computing Machinery, Vol. 14 No. 4 (1971) pp 221-227.

[WIRT73] Wirth, N., "On the Composition of Well-Structured Programs," Computing Surveys, Vol. 6 No. 4 (1974) pp 247-259.

[YOUR75] Yourdon, E., Techniques of Program Structure and Design, Prentice-Hall (Englewood Cliffs, NJ) 1975.

APPENDIX D
ON REDUCING AMBIGUITIES IN METHODOLOGY DEFINITIONS

INTRODUCTION

Efforts to enhance the preciseness of design methodology descriptions have been following two main directions. One avenue being pursued involves attempts to describe formally the nature of the design specifications and the logical and performance evaluation criteria used in determining the acceptability of proposed designs [ALFO79]. The second direction being observed, primarily in the data processing area [BERG81], is characterized by efforts to treat methodologies as well-defined algorithms. While the notion of reducing design to a mechanical procedure may be subject to debate, its contribution to promoting precise definitions of the methodologies is beyond dispute. Unfortunately, despite the trend toward better defined methodologies, their presentation continues to be almost exclusively informal and, as a consequence of this fact, it is plagued by ambiguities.

The work being reported here is based on the premise that specification languages have an important role to play in the generation of unambiguous methodology definitions which, in turn, affect the way in which configuration control and project planning will be carried out in the computer-aided design systems of the future. Precise methodology definitions hold the promise for better communication among designers and also the key to increasing the designer's capacity to study, understand, evaluate, and compare one methodology against another. Furthermore, the inclusion of the methodology specification as part of the database of a computer-aided design system opens the possibility for a better enforcement of the correct use of methodology on a given project. Its use as an input to the project management tools is also being contemplated.

These opportunities are only now beginning to be explored and no similar efforts have been yet reported in the open literature, to the best of our knowledge. This paper reports the results of an investigation whose three major objectives, while falling short of the stated goals, represent a necessary stepping stone toward them. The first objective is to identify a way of describing the structure of and the relations between various products of the design process. They are referred here as configuration items and may include such things as system requirements, program specifications, module design, code, etc. The second objective is the ability to capture changes in the state of these configuration items and to define consistency constraints between their states. The third major objective is to be able to prescribe the sequencing of design activities permitted by the methodology in question.

The next four sections of the paper describe a proposal for a methodology definition language and illustrate it on a variation of a well-known methodology, top-down program design. A separate section is dedicated to discussing the way in which the issue of backtracking due to design errors is addressed in this paper. The concluding section summarizes the experience to date with the methodology definition proposal and some of difficulties encountered.

OVERVIEW OF THE METHODOLOGY DEFINITION APPROACH

The methodology definition consists of three parts: the definition of the configuration items, the definition of consistency constraints, and the sequencing of design activities. Syntactically, the definition assumes the following appearance with the keywords being capitalized. (See also Figure D-1.)

METHODOLOGY methodology-name.

CONFIGURATION ITEMS.

...

CONSISTENCY CONSTRAINTS.

...

sequencing of design activities

MEND.

The configuration items represent entities being generated and used in the design process, e.g., documentation, programs, hardware components, etc. The exact configuration items one may include in the definition of the methodology depends upon the nature of the methodology and upon the granularity of its description. Program modules, for instance, may be relevant for a program design methodology, but they may not appear in a software system design methodology which treats programs as the lowest level entities of interest to the designer. Aside from configuration items identification, the methodology definition needs to include the structural relations between these items. Considering the case of the program modules again, they are grouped in a hierarchical structure to form a program. Moreover, the program, in turn, has a program specification (another configuration item) and maybe a program design document. All this information has to be stated in the configuration items definition. As shown in the next section this is accomplished through the use of a method borrowed from Hoare's treatment of recursive data structures [DAHL72]. Because of the use of recursive data structures, the definition of the consistency constraints employs a LISP-like notation for defining the structural and state invariants over the configuration items [WEIS67]. It is explained two sections later.

The consistency constraints over the configuration items originate in the design rules prescribed by the methodology and reflect properties that remain invariant throughout the design process. (They are not unlike the consistency constraints present in a database.) Because the design rules are prescribed by the sequencing of design activities, the consistency constraints may appear to be unnecessary unless one requires a certain level of redundancy in the definition. (Redundancy is considered desirable and forms the basis for the self-consistency of the specification.) However, the presence of the consistency constraints is necessary and desirable from another point of view. Since, as shown later, many of the design activities are presented informally (natural language), the consistency constraints enable one to reduce the ambiguity level intrinsic to natural language. Furthermore, certain possible but

undesirable sequences of design activities may be ruled out. (This situation occurs when using nondeterministic constructs in the activity sequencing part of the language.)

The sequencing of design activities in a methodology is not, generally speaking, different from the sequencing of instructions in programming languages. Consequently, most control abstractions (i.e., flow of control constructs) have been borrowed from structured languages, with some modifications. By necessity, they include sequential type constructs, parallel type constructs, nondeterminism and recursion. The last three require some discussion. The need for high degrees of concurrency in the methodology definition is motivated by the fact that most projects involve designers that work in parallel on different aspects of a problem. Nondeterminism is required for two key reasons. The first one is the frequent occurrence of situations where the designer has to choose among several courses of action based on personal experience and methodology supplied guidelines rather than algorithmically. The second reason is the equally common situation where the methodology is only partially defined and still under development and evaluation. Finally, regarding recursion, it is required by the use of recursive data structures.

CONFIGURATION ITEMS DEFINITION

The simplest configuration item is one which has no recognized structure. It is called an atom. Given any number of configurations items, they may be used to create more complex configuration items: sets, which are abstractions of collections of items, and recursive structures which render the organization of documents or actual products. The BNF specification of the syntax used in specifying the configuration items looks as follows:

```
<configuration-structure>
 ::= <item-definition> ";" |
    <item-definition> ";" <configuration-structure>

<item-definition>
 ::= <item-name> "=" "(" <item-tuple> ")" |
    <item-name> "=" "{" <item-list> "}"

<item-tuple>
 ::= <atom> | <item-name> |
    "SEQUENCE" <item-name> |
    <item-tuple> "," <item-tuple>

<item-list>
 ::= <atom> | <atom> "," <item-list>
```

In order to better understand the approach, let us consider the relatively simple case of a top-down program design methodology. The intent is to carry out the example through completion by starting " here with the definition of the configuration items and continuing it in the

next two sections. A reasonable set of configuration items one might consider is bound to include the original program specification, the program design, and the code. The program specification may include the input and output assertions and some sample test data. The program design generally consists of the program data structures and the module definitions. The modules form a hierarchical structure which is isomorphic to that formed by the subroutines present in the program code. Keeping all these in mind, one may build the following configuration items definition:

METHODOLOGY top-down-design.

CONFIGURATION ITEMS.

```
program-specification
    = (input-assertion, output-assertion, test-data);

program-design
    = (data-structures, module);

module
    = (module-name, module-definition, SEQUENCE module);

program-code
    = (subroutine);

subroutine
    = (subroutine-name, subroutine-code, SEQUENCE subroutine);
```

CONSISTENCY CONSTRAINTS.

...

sequencing of design activities

MEND.

(Note: The definition above may appear to rule out programs whose structures are not trees. Actually, it is shown in the next section that this is not so.)

CONSISTENCY CONSTRAINTS SPECIFICATION

Most often, the consistency constraints one may want to specify involve more than mere structural properties of the configuration items. Take, for instance, the relation between modules and subroutines. They form two isomorphic structures, i.e., for each module there is a corresponding subroutine and the subroutines called by it correspond to the modules called by the said module. This relation, however, is not an invariant over the configuration items because it holds true only at the end of the design process and not throughout. One way to assist the designer in the formulation of consistency constraints such as this is through the introduction of the concept of state. States and state transitions may be included in the consistency constraints definition

AD-A126 101

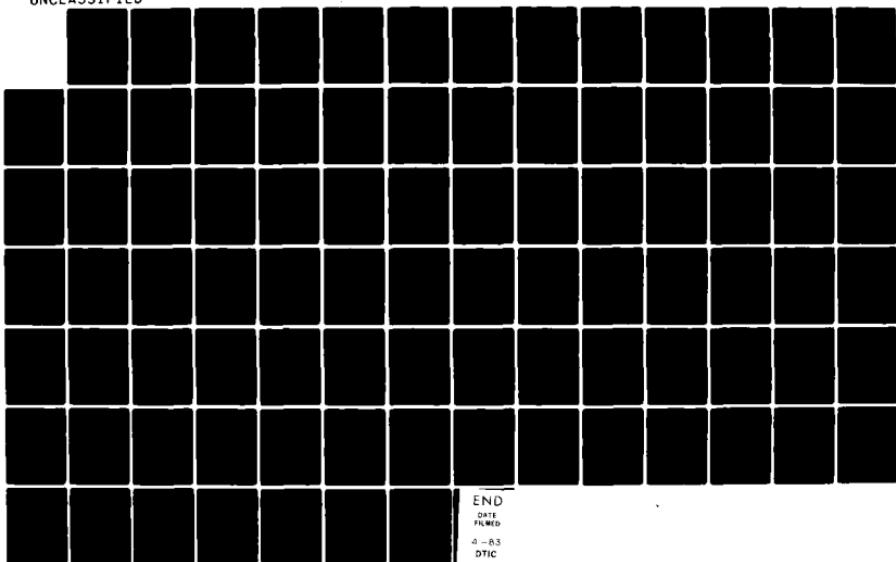
TOTAL SYSTEM DESIGN (TSD) METHODOLOGY ASSESSMENT (U)
WASHINGTON UNIV SEATTLE DEPT OF COMPUTER SCIENCE
G ROMAN ET AL. JAN 83 RADC-TR-82-331 F30602-80-C-0284

4/4

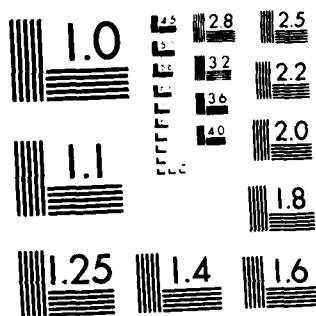
NL

UNCLASSIFIED

F/G 9/2



END
DATE
FILED
4-83
DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

next two sections. A reasonable set of configuration items one might consider is bound to include the original program specification, the program design, and the code. The program specification may include the input and output assertions and some sample test data. The program design generally consists of the program data structures and the module definitions. The modules form a hierarchical structure which is isomorphic to that formed by the subroutines present in the program code. Keeping all these in mind, one may build the following configuration items definition:

METHODOLOGY top-down-design.

CONFIGURATION ITEMS.

```
program-specification
    = (input-assertion, output-assertion, test-data);

program-design
    = (data-structures, module);

module
    = (module-name, module-definition, SEQUENCE module);

program-code
    = (subroutine);

subroutine
    = (subroutine-name, subroutine-code, SEQUENCE subroutine);
```

CONSISTENCY CONSTRAINTS.

...

sequencing of design activities

MEND.

(Note: The definition above may appear to rule out programs whose structures are not trees. Actually, it is shown in the next section that this is not so.)

CONSISTENCY CONSTRAINTS SPECIFICATION

Most often, the consistency constraints one may want to specify involve more than mere structural properties of the configuration items. Take, for instance, the relation between modules and subroutines. They form two isomorphic structures, i.e., for each module there is a corresponding subroutine and the subroutines called by it correspond to the modules called by the said module. This relation, however, is not an invariant over the configuration items because it holds true only at the end of the design process and not throughout. One way to assist the designer in the formulation of consistency constraints such as this is through the introduction of the concept of state. States and state transitions may be included in the consistency constraints definition

section and referred in the statement of other invariants. The relation above could thus be reformulated by adding the condition that both the program-design and the program-code are in the state called "frozen." The notion of state also helps capture the progress made by the designer.

The syntax employed in the state specification is given below.

```
<state-assignment>

 ::= <item-name> ":" <initial-state> ";" |
 <item-name> ":" <initial-state> "," <transitions> ";" |
 <atom> ":" <initial-state> ";" |
 <atom> ":" <initial-state> "," <transitions> ";"

<transitions>

 ::= <state> "-->" <state> |
 <transitions> "," <transitions>
```

Assisted by this notation system, the following state definitions may be added to the example.

METHODOLOGY top-down-design.

CONFIGURATION ITEMS.

...

CONSISTENCY CONSTRAINTS.

STATES.

```
program-specification: given;
program-design: not-started,
                not-started --> in-progress,
                in-progress --> frozen;
data-structures: null,
                null --> designed;
module:          null,
                null --> designed;
program-code:   not-started,
                not-started --> in-progress,
                in-progress --> frozen;
subroutine:     null,
                null --> stubbed
                stubbed --> coded
                coded --> tested;
```

INVARIANTS.

...

sequencing of design activities

MEND.

The approach to invariants definition is illustrated by constructing two invariants required by the example. Some knowledge of LISP is assumed on the part of the reader. A more convenient notation is being planned but its introduction would increase the size of the paper without adding to its technical contents. A configuration item name followed by a state name in square brackets should be treated as a predicate that evaluates to true if the item is in the specified state, and to nil, otherwise.

The first invariant states that the coding may not start until the completion of the design:

```
(COND ( program-code[in-progress]  program-design[frozen])
      ( T T) )
```

The second invariant originates in the requirement that the design may not be frozen unless all modules have been designed:

```
(COND (program-design[frozen]
      (AND data-structures[designed]
           (NIL ((check LAMBDA (X)
                           (COND ((NIL X) NIL)
                                 (X[designed]
                                   ((checkseq LAMBDA (Y)
                                       (COND ((check (CAR Y)) T)
                                             (T (checkseq (CDR Y))))
                                       ) (CDDR X)))
                                   (T T)
                                 )
                               ) (CADR program-design))
                           )
                         )
           (T T))
```

In the above invariant the function check performs a depth-first search of the module call tree and returns T if and only if a module which has not been designed is encountered. This search is performed recursively through the checkseq function which invokes check for each of the modules called by a given module.

In a similar manner, all other invariants may be constructed. A complete specification would require at least two more invariants. The first one is needed to establish the isomorphism between the structure of the modules from the program design and the structure of the subroutines present in the code. The second has a more subtle flavor and states the fact that two modules bearing the same name must be the identical. This constraint is an artifact of the fact that directed acyclic graphs are represented as trees by duplicating certain of the graph nodes.

SEQUENCING OF DESIGN ACTIVITIES

The main body of the methodology is described by a combination of formal and informal statements sequenced by means similar to those employed by programming languages. (See Figure D-2.) Tasks, subtasks, and procedures are subject to the same scoping and invocation rules as procedures in block-structured languages. If there are small differences, they are due to the need to capture some concepts peculiar to methodologies. Both tasks and subtasks, for instance, have a section dedicated to project review activities. The section is entered just before the returning from the task or subtask. Another distinction is the fact that a task may not be invoked recursively or from other tasks, subtasks, or procedures. The motivation for this limitation stems from the intent that tasks be used to describe major design baselines.

Before continuing the example, one unusual feature of the language must be pointed out in order to avoid possible confusion. It concerns the place of definition for tasks, subtasks, and procedures. The definitions always appear at the place of first mention for the respective task, subtask, or procedure. The choice has been made based on the results of early experiments in the use of the language which indicated that the placement of the definitions at any other place distracts the reader who would encounter definitions prior to understanding their role in the description or would have to search for definitions when the first mention of some task, subtask, or procedure is made in the text. It is felt, however, that in the future both in-line and separate definition capabilities may have to be provided, particularly for use with very large descriptions.

The program design methodology used for illustration purposes includes two phases: program design and coding. Because coding should not be started before the completion and the review of the design, the two phases may be separated into two distinct tasks.

METHODOLOGY top-down-design.

CONFIGURATION ITEMS.

...

CONSISTENCY CONSTRAINTS.

...

TASK design.

...

TREVIEW.

...

TEND.

TASK code.

...

TREVIEW.

...

TEND.

MEND.

The program design starts with the tentative selection of the program data structures. After designing the top-level module, the design proceeds with the design of the modules identified in the definition of the top-level module. A strict one level at a time strategy is followed until no more modules are identified. Appropriate checks and adjustments take place throughout the design process rendered in the following definition of the program design task. (The state changes have been omitted from the definition.)

TASK design.

Design data-structures.
Design top-level module.

SUBTASK level-design(x='top-level module').
Identify modules called by x.
FOR z IN modules called by x DO
{IF z needs to be refined
THEN {Design z.
Data-structures changed => BACK design.
F(Verify consistency of z with x) => BACK}}
STREVIEW.
F(Verify refinement of x.) => BACK level-design(x).
FOR z IN modules called by x DO { // level-design(z).}
STEND.

TREVIEW.

F(Verify program-design against program-specification.)
=> BACK design.
Declare program-design frozen.
TEND.

In contrast with the program design, coding is assumed to follow a top-down direction but not in the strict level by level manner. The designer has the freedom to guide the coding and testing based on testing dependencies, as long as testing and coding are not done separately, but progress together. As a way to restrict the designer from coding too much before attempting testing, no more than five untested subroutines are permitted to exist at one time. The same simple style free of the formal state transitions is used to describe also the coding task.

TASK coding.

Code the main program and stub all subroutines it calls.
Debug main using stubs.
LOOP
{(All subroutine are tested.) => BREAK.
IF fewer than 5 subroutines are untested
THEN { T => Replace one stub by actual code.
T => Debug available code.}
ELSE Debug available code.

TREVIEW.

F(Check agreement between code and the design.) => BACK.
F(Obtain user acceptance of the program.)

```
=> { T => BACK design. |
      T => BACK coding. }
TEND.
```

The last thing to be done is to establish the entry points for program maintenance activities and the rules by which the appropriate entry point is selected.

METHODOLOGY top-down-design.

CONFIGURATION ITEMS.

...

CONSISTENCY CONSTRAINTS.

...

ENTRY major-maintenance.

Design errors and major enhancements.
END.

TASK design.

...

TREVIEW.

...

TEND.

ENTRY minor-maintenance.

Changes to the printout format and coding level errors.
END.

TASK code.

...

TREVIEW.

...

TEND.

MEND.

The semantics of an entry point is similar to that of backtracking to be discussed in the next section. The only difference lies in their different causes. One is due to the check that takes place in the development of the program (backtracking), while the other has its roots in either the decision to enhance the program or the discovery of errors during production.

BACKTRACKING

The discovery of design errors, the identification of a better design path, changes in the specifications, and a newly acquired understanding of the technological impact of a proposed design are some of the most common reasons for backtracking. One can hardly conceive of a methodology that denies the possibility for backtracking to occur. Furthermore, while

intentional neglect of backtracking for the sake of simplifying the presentation of some methodology is common practice, to disregard its effects is not acceptable. The cost of backtracking is an important factor in selecting one methodology over another. Methodology design is centered around cutting the cost of backtracking through automated and non-automated checks, through exercising control over the degree of backtracking being permitted, through automatic detection of the potential side effects of backtracking, etc.

The costs of backtracking, however, needs to be weighed against that of the checks employed for the sake of reducing it. In general, increased degrees of automation enable one to enjoy both frequent design checks which reduce backtracking due to errors and greater opportunities for the exploration of the design space. The cost of backtracking is also affected by the project management procedures. For instance, when several designers work in parallel any backtracking that impacts more than one of the designers may prove very costly compared with backtracking concerning one of them alone.

Such considerations strongly suggest that a methodology definition approach ought to provide a mechanism for explicit structuring of the backtracking process and should lay the foundation for employing (methodology-based) quantitative project planning methods. These methods would require knowledge of the backtracking patterns, backtracking probabilities due to various causes, and the cost associated with various backtracking procedures. By placing explicit backtracking statements, the approach put forth in this paper enables the definition of backtracking patterns, but their use in project planning will have to be explored later on. The adoption of explicit backtracking statements (e.g., BACK name.), however, raises two issues concerning their impact on the flow of control and on the configuration items that have been generated prior to executing the backtracking statement.

The first issue is rather easy to resolve. The name that follows the backtracking command defines the backtracking point and may be the name of some group of statements or the name of some task, subtask, or procedure. In the first case, BACK could be treated as a "goto" to a label which is defined in the current referencing environment and which is always encountered prior to the execution of the BACK command. (This condition may be checked statically by means of data flow analysis.) For instance,

```
..
label: {...}
..
BACK label.
..
IF ... THEN {...} ELSE {... BACK label. ...}
..
```

represents a correct use of the BACK command because the statement named "label" is always executed prior to the "BACK label." statement. The following sequence of statements, however, is improper.

```
IF ... THEN label: {...}
..
BACK label.
..
IF ... THEN {...} ELSE {... BACK label. ...}
..
```

The rule may be easily extended to procedures, tasks and subtasks by considering their names in place of the statement label. However, only tasks, subtasks, and procedures which have not been exited yet may be backtracked. Furthermore, sequences of recursive invocations are backtracked up to the first invocation of the named subtask or procedure.

The backing up of the flow of control must be accompanied by a corresponding backtracking of the configuration items state. It is easy to conceive that the state of the design is saved at every backtracking point and reestablished any time backtracking brings the flow of control back to the respective point. This way of looking at backtracking has one drawback. It seems to suggest that all the design generated prior to backtracking is lost together with the reason for the backtracking itself. Therefore, the interpretation adopted in this paper requires all results generated after encountering the backtracking point to be tagged as needing the designer's revalidation. The designer may chose to accept some results the way they are, to discard others, and to alter still other configuration items, all decisions being based on knowledge of the backtracking cause. Thus, no design decisions potentially affected by backtracking are left unchecked.

CONCLUSIONS

The methodology definition approach proposed in this paper has been used in the specification of several methodologies. Their descriptions, which vary in size from one to five single-spaced pages, have confirmed some of the advantages that were expected. Its use on a methodology development project has yielded significant quality improvements in the communication between the members of the research team. Many problems that were overlooked in the informal presentations of new proposals for a distributed systems design strategy have been rapidly uncovered during the effort of formally describing the methodology.

Despite the early successes in using the methodology definition approach, much work still lies ahead. There are four areas that seem to require immediate attention. First, it is the issue of incorporating the approach in some computer-aided design system for purposes of methodology enforcement and project planning. Second, project planning techniques based on formal methodology definitions need to be developed and parameterized so as to be usable over a large class of problems and by a variety of organizations. Third, more practical experience is required in using the approach. Of particular interest are methodologies characterized by high degree of backtracking and concurrency. Finally, the availability of a formal definition offers the possibility to carry out quantitative evaluations regarding optimal placement and frequency of

various design checks within methodologies. Preliminary work strongly indicates that these future research directions hold great promise for significant system design productivity payoffs.

REFERENCES

- [ALF079] Alford, M., "Requirements for Distributed Data Processing Design," Proc. 1'st Int. Conf. on Distributed Computing Systems, pp. 1-14, October 1979.
- [BERG81] Bergland, G. D., "A Guided Tour of Program Design Methodologies," Computer 14, No. 10, pp. 13-37, October 1981.
- [DAHL72] Dahl, O.-J., Dijkstra, E. W., and Hoare, C. A. R., Structured Programming, Academic Press, 1978.
- [WEIS67] Weissman, C., LISP 1.5 Primer, Dickenson Pub. Co., 1967.

FIGURE D-1: METHODOLOGY SPECIFICATION STRUCTURE.

METHODOLOGY DEFINITION

 CONFIGURATION ITEMS

 CONSISTENCY CONSTRAINTS

 SEQUENCE OF TASKS AND MAINTENANCE ENTRY POINTS

 ENTRY

 | ENTRY POINT CONDITIONS

 TASK

 DESIGN/ANALYTIC ACTIVITIES, ENTRY POINTS AND
 INVOCATIONS OF SUBTASKS AND PROCEDURES

 SUBTASK

 | DESIGN/ANALYTIC ACTIVITIES, ENTRY POINTS AND
 | INVOCATIONS OF SUBTASKS AND PROCEDURES

 | PROCEDURE

 | DESIGN/ANALYTIC ACTIVITIES, ENTRY POINTS AND
 | INVOCATIONS OF SUBTASKS AND PROCEDURES

 | SUBTASK COMPLETION REVIEW

 | ANALYTIC ACTIVITIES

 | AND INVOCATIONS OF SUBTASKS AND PROCEDURES

 | TASK COMPLETION REVIEW

 | ANALYTIC ACTIVITIES

 | AND INVOCATIONS OF SUBTASKS AND PROCEDURES

FIGURE D-2: SEQUENCING OF DESIGN ACTIVITIES.

TASK name(parameters).	SUBTASK name(parameters).
..	...
TREVIEW.	STREVIEW.
..	...
TEND.	STEND.
PROC name(parameters).	ENTRY name.
..	...
PEND.	END.
activity	informal description followed by a period list of state transition rules (e.g., item[s1-->s2, s3-->s4].) new state assignment (e.g., item[s1].) invocation of task, subtask, or procedure (e.g., INVOKE p.)
condition	activity failure (e.g., F(activity)); activity success (e.g., S(activity)); formal and informal predicates; state test (e.g., item[s1]).
sequence	a b ... c
if-then-else	IF condition THEN a ELSE b
if-then	condition => a
	IF condition THEN a
group	name: { a b ... c }
	(Note: it acts as a DO group in PL/I.)
parallel group	name: { a // b // ... // c }
	(Note: it acts as a COBEGIN-COEND block.)
nondeterminism	name: { condition => a ... }
	(Note: one activity preceded by a true condition is selected as desired by the designer.)
iteration	name: LOOP a
	(Note: 'BREAK loop-name.' and 'NEXT loop-name.' are used to exit the loop and to go back to its beginning, respectively.)

FIGURE D-2 (cont.): SEQUENCING OF DESIGN ACTIVITIES.

sequential for name: FOR item IN item-list DO a
(Note: activity a is repeated for each item in order.)

parallel for name: FOR item IN item-list DO { // a}
(Note: activity a is carried out for each item
in parallel.)

backtracking BACK name.
(Note: the name may be a construct label or the
name of a procedure, task, or subtask from
the calling sequence; when no label is provided,
the most recent invocation of a procedure, task,
or subtask is restarted.)

others RETURN.
(Note: normal return from procedures.)

DONE.
(Note: normal return from tasks, and subtasks;
the task/subtask reviews are not omitted.)

ABORT name.
(Note: failure return from procedures, tasks, and
subtasks; any procedure, task, and subtask
in the calling sequence may be aborted.)

APPENDIX E
A FORMAL TREATMENT OF DISTRIBUTED SYSTEMS DESIGN

E.1 INTRODUCTION

The system design stage covers all design activities involved in taking a set of system design requirements and generating the specification of the hardware and software requirements for the respective system. There are two phases that make up this stage: the system architecture design phase and the system binding phase. The former deals with the selection of an overall system architecture which accomplishes the intended system functionality and which, under a reasonable set of technological assumptions, meets the performance and other constraints originating with the system requirements. The proposed architecture and all the design decisions taken during this phase form a processing model used as input to the binding phase.

The binding phase, based on the limited degrees of freedom still left open by the system architecture design phase and based on market availability, identifies a particular mix of software and hardware and produces specifications for all needed components. The nature of the specifications, however, may vary from component to component depending on its intended realization (software or hardware) and on the manner in which it is to be obtained (off-the-shelf, through customization, or custom-made). The system design stage is also concerned with the integration of the system components from the point when both the software and the hardware components are available and up to the point when the system is offered for customer acceptance testing.

In order to carry out the tasks of the system design stage, a set of specifications languages is necessary to formally characterize the design at different points in the design process. The particular specifications needed during this stage are the system requirements, which serve as input to the system architecture design phase; the processing model, which is the interface between the system architecture design phase and the system binding phase; and the hardware/software requirements, which is the output of the system binding phase. Together these specification media provide the core around which a system design methodology can be organized.

The purpose of this appendix is to define a formal model of each of the specifications that are required by the system design stage, as well as to define the interrelationships between these specifications. The first definition is that of the system requirements model, which is presented in section E.2. Next, the processing model is defined in section E.3 along with its relationship to the system requirements. Finally, in section E.4 the hardware/software requirements model is defined along with its relationship to the processing model.

Note: In this appendix the following notation will be used.

(!A x) - For all x :

(?E x) - There exists an x such that :

E.2 SYSTEM REQUIREMENTS DEFINITION

The system requirements are a description of the functional requirements for the proposed design and the constraints which the design must meet. For the requirements to be analyzable, some structure must be imposed upon them. The following definitions are intended both to formalize the concept of system requirements definition and to impose a hierarchical structure on that definition in order to support the notion of stepwise decomposition inherent in the TSD philosophy. First, a definition of the concept of system requirements is presented in section E.2.1. This is followed in section E.2.2 by a more detailed definition of one component of the system requirements called the conceptual model. Finally, the decomposition of a conceptual model in a hierarchical manner is defined and discussed in section E.2.3.

E.2.1 SYSTEM REQUIREMENTS

$$SRQ = (CM, R, CQ, eval)$$

CM - conceptual model
R - domain of possible system realizations
CQ - set of design constraints
eval - system evaluation procedure

The system requirements (SRQ) are defined as a 4-tuple consisting of a conceptual model, a set of possible system realizations, a set of constraints on the system, and a system evaluation function. The conceptual model (CM), which is formally defined in the next section, is a model of the functionality of the system with respect to its environment. The conceptual model does not incorporate any non-functional constraints such as speed or size limitations. This allows for formal treatment of system functions independent of the complexity of performance requirements. The set of possible system realizations (R) is a conceptual entity consisting of all possible system realizations which exhibit the functionality defined in the conceptual model. Although this set is not necessary for the definition of the system requirements, the concept it represents will be used later in the system design definition. The set of constraints (CQ) consists of all explicit or implied constraints on the performance, packaging, implementation, etc. of the system that exist independent of the system design process. The constraints together with the conceptual model fully specify the system to be designed as the conceptual model determines the set of possible realizations and the constraints determine a (possible empty) subset of those realizations that meet all of the non-functional requirements. The determination of the subset of realizations which meet all of the requirements is done by the system evaluation function (eval). More specifically, given a set of realizations R and a set of constraints C, eval(R,C) is a set containing only those members of R which are consistent with the requirements C.

E.2.2 CONCEPTUAL MODEL

$$CM = (ES, ESO, SS, SSO, F)$$

ES - set of environmental states
ES0 - set of initial environment states
SS - set of system states
SS0 - set of initial system states
F - functionality
 $F \subseteq ((ES \times SS) \times (ES \times SS))$

The conceptual model (CM) is defined as a 5-tuple consisting of a set of environmental states (ES), a set of initial states for the environment (ES0), a set of system states (SS), a set of initial states for the system (SS0), and a transition mapping (F). The sets ES and ES0, where ES0 is a subset of ES, are used to model the environment of the system, as the allowable functionality of the environment must be determined in order to establish the functionality of the system. Similarly, the sets SS and SS0, where SS0 is a subset of SS, are used to model the system. Given this static characterization of the system and its environment the mapping F is used to model their dynamic properties. F can be described as a subset of $((ES \times SS) \times (ES \times SS))$ where the first element in the pair can be thought of as the current state of the system and the environment, and the second element as the successor state. Since many to many mappings in F are allowed, it is possible to model non-deterministic state transitions.

The thrust of these definitions is that the system functionality can be modeled by a set consisting of all valid sequences of system-environment states. Such a set represents only the possible orderings of occurrences in the system, and implies nothing of the timing involved. Under the CM definition a valid sequence can be characterized as follows. A sequence $((e_1, s_1), (e_2, s_2), \dots, (e_n, s_n), \dots)$ is valid iff e_1 is a member of ES_0 , s_1 is a member of SS_0 , and each pair of successive states $((e_i, s_i), (e_{i+1}, s_{i+1}))$ is a member of F.

E.2.3 DECOMPOSITION

Since the concept of hierarchical top-down refinement is central to the TSD philosophy, a hierarchical refinement of conceptual models is a natural method of construction under this philosophy. Formally, a conceptual model can be defined in terms of a hierarchy of models $(CM_1, CM_2, \dots, CM_n)$ where each CM_{i+1} is a decomposition of CM_i . The decomposition relationship, denoted $CM' \text{ REFINES } CM$, is defined as follows:

Given $CM = (ES, ESO, SS, SSO, F)$ and $CM' = (ES', ESO', SS', SSO', F')$,
then $CM' \text{ REFINES } CM$ iff there exists a function

$$\text{PHI} : (ES' \times SS') \rightarrow (ES \times SS)$$

For every initial state (e_0', s_0') there exists an initial state (e_0, s_0) such that

$\text{PHI}(e_0', s_0') = (e_0, s_0)$

PHI is onto $(\text{ES} \times \text{SS})$

$((e_1', s_1'), (e_2', s_2')) \in F'$
AND $\text{PHI}(e_1', s_1') = (e_1, s_1)$
AND $\text{PHI}(e_2', s_2') = (e_2, s_2)$
AND $\text{NOT } (e_1, s_1) = (e_2, s_2)$
IMPLIES $((e_1, s_1), (e_2, s_2)) \in F$

In English, CM' REFINES CM iff each state in CM' can be mapped to a unique state in CM , all states in CM are covered by the mapping, all initial states in CM' map to initial states in CM , and all allowable transitions in CM' correspond either to no transition in CM or an allowable transition in CM . Aside from its use in the definition of a hierarchy of conceptual models, REFINES can be used to define equivalence as follows:

CM_1 is equivalent to CM_2 IFF
 CM_1 REFINES CM_2
AND CM_2 REFINES CM_1

E.3 PROCESSING MODEL DEFINITION

The methodology described in section 3.4 models systems as a hierarchy of related design specifications where the specification at one level reveals a design solution for some problem which is formally defined within the level above. The processing model reflects this view by characterizing the system as a total order over a finite set of design specifications. Although the total ordering is not really necessary, it has been adopted in order to simplify the presentation of both the processing model and the system design strategy. Furthermore, the extrapolation to an upside down tree (a tree in which the level of each node is the longest distance from a leaf rather than the root) is trivial. A formal definition of this processing model is presented in the next section, followed by a definition of the system design specifications which make up the processing model in section E.3.2. The relationships which must hold between the processing model and the system requirements, as well as between different design specifications in the processing model are defined in section E.3.3.

E.3.1 DEFINITION OF THE PROCESSING MODEL

PM = (DS1, DS2, ... DS_n)

DS₁ - System design specifications
DS₁ IMPLEMENTS SRQ
DS_{i+1} SUPPORTS DS_i for i<n
PM SBOUND.TO MCH IFF (DS_n SPECIFIES MCH)

The processing model is defined as a linear order of design specifications which meet a number of criteria. First, the initial design specification (DS₁) must implement the system requirements (SRQ). The IMPLEMENTS relationship, formally defined in section E.3.3, essentially requires that the design specification implements the functionality established by the conceptual model and that at least one realization of that design specification meets the system constraints. The next requirement is that each successive design specification must support its immediate predecessor. The SUPPORTS relationship requires that one design specification implements the processor structure portion of another design specification (see section E.3.3). The last criterion for the processing model is one of completeness: the model completely specifies a system architecture when it can be superficially bound to a network of physical machines and support processes. A processing model PM is superficially bound to a net of machines MCH if and only if the lowest level design specification DS_n SPECIFIES MCH. The SPECIFIES relationship is defined more formally in section E.3.3, but essentially states that the processor structure of DS_n can be mapped directly onto MCH.

E.3.2 DEFINITION OF THE SYSTEM DESIGN SPECIFICATION

DS = (PSS, PRS, ALC, PFS, BD, C, eval)

PSS - process structure
PRS - processor structure
ALC - process/processor allocation
PFS - performance specifications
BD - binding options
C - constraints
eval - system evaluation procedure w.r.t. C

A system design specification is a 7-tuple consisting of a process structure, a processor structure, an allocation of processes to processors, a set of performance specifications, a set of binding options, a set of system constraints, and a procedure for evaluating systems with respect to the constraints. The process structure is a network of processes and conceptual communications links, and essentially describes the functional elements which interact to carry out the overall system function. The processor structure describes a network of conceptual processors and their interconnections which provides the support structure on which the process structure resides. The process/processor allocation defines a mapping of processes and inter-process communications in the process structure onto the processors and interconnections in the processor structure. The performance specifications attach performance requirements to the process and processor structures and also contain performance data derived from these structures that can be used in the design validation process. The binding options represent a conceptual set of feasible realizations of the system design which meet the binding constraints (to be defined later). The set of constraints consists of the constraints established by the system requirements. Finally, the system evaluation function serves to define the set of binding options by evaluating the validity of potential system realizations with respect to a set of constraints. A more formal definition of each of these components of the design specification is presented in the following subsections.

E.3.2.1 PROCESS STRUCTURE

PSS = (PS, LK)

PS - set of processes
PS = { P ; P=(SP, sp0, TP) }
SP - set of process states
sp0 - set of initial process states
TP - state transition rule
TP SUBSET.OF (SP x SP)

LK - set of links between processes
LK = { L ; L=(PL, SL, s10, TL) }
PL - processes linked by L
PL SUBSET.OF PS
SL - states internal to the link

```

s10 - set of initial link states
TL - link communication protocol
    TL SUBSET.OF (lcstates x lcstates)
    where
        lcstates = (SL x SP1 x ... SPn)
        Pi MEMBER.OF PL
        Pi = (SPi, sp0i, TPi)

```

The process structure (PSS) is defined as an ordered pair (PS,LK), where PS is a set of processes and LK is a set of links. Processes are the individual functional entities in the system, and they are defined more fully below. Links are the logical communications paths of the system, and can be viewed as the medium by which inter-process message passing is carried out. The functionality of the entire system is determined by the superposition of these two sub-specifications of the system.

A process P is defined as a triple (SP, sp0, TP), where SP is a set of process states, sp0 is a set of initial process states, and TP is a set of valid transitions from one process state to another. The functionality of the process is the set of all valid sequences of states, which can be determined from sp0 through successive application of the transitions in TP (the conceptual model was defined similarly). The set SP describes not only the internal states of the process, but also those of the process interfaces to the links. Hence, the interaction of the system and its environment is specified completely through this scheme.

A link L is defined as a 4-tuple (PL, SL, s10, TL), where PL is a set of processes, SL is a set of internal link states, s10 is a set of initial states, and TL is a set of valid transitions which serves to specify the link protocol. PL is a subset of PS, and represents those processes which can communicate through the link subject to the link protocol. The set SL describes the internal states of the link, but does not include knowledge of any portion of the internal process state. The functionality of the link and its interaction with the processes it interconnects is specified by the state transition rules in TL, which specify transitions from one aggregate process/link state to another. Hence, the functionality of the links is intimately bound to that of the processes it interconnects.

E.3.2.2 PROCESSOR STRUCTURE

PRS = (PR, IC)

PR - set of processors
 PR = { Q | Q=(SQ, sq0, TQ) }
 SQ - set of processor states
 sq0 - set of initial processor states
 TQ - state transition rule
 TQ SUBSET.OF (SQ x SQ)

IC - set of processor interconnections
 IC = { W | W=(PW, SW, sw0, TW) }
 PW - processors being interconnected

SW - set of internal interconnection states
 sw0 - set of initial interconnection states
 TW - interconnection protocol
 TW SUBSET.OF (icstates x icstates)
 where
 icstate = (SW x SQ1 x ... SQn)
 Qi MEMBER.OF PW
 Qi = (SQi, sq0i, TQi)

The processor structure (PRS) is defined as an ordered pair (PR, IC), where PR is a set of processors and IC is a set of processor interconnections. PR is a set of conceptual processors, and represent the functional processing elements in the system. IC is a set of processor interconnections, which defines the communications paths between processors. The network of processors and interconnections so described provides the support structure upon which the process structure resides (in a manner like that of a virtual machine supporting the execution of a user process).

Each processor Q in PR is defined as a triplet (SQ, sq0, TQ), where SQ is a set of processor states, sq0 is a set of initial processor states, and TQ is a set of valid state transitions. The processor states include both the internal processor state description and the state of the interfaces to the communications interconnections for that processor. Thus, in a manner similar to that of the process definition, the functionality of the processor with respect to its environment is defined.

An interconnection W in IC is defined as a 4-tuple (PW, SW, sw0, TW), where PW is a set of processors, SW is a set of internal interconnection states, sw0 is a set of initial interconnection states, and TW is a set of valid interconnection state transitions. PW is of course a subset of PR and represents those processors which can communicate across the interconnection subject to the interconnection protocol. SW models the internal state of the interconnection but does not include knowledge of any portion of the processor states. The functionality of the interconnection and its interaction with the processors it connects is specified by TW, which is a set of allowable transitions from one aggregate processor/interconnection state to another. Hence, the functionality of the interconnection is intimately bound to that of the processors it interconnects.

E.3.2.3 PROCESS/PROCESSOR ALLOCATION

ALC = (AF, af0, TA)

AF - set of all allocations functions
 AF = {A : A : prstates --> psstates}
 prstates - composite processor structure states
 prstates = SQ1 x SQ2 x ... SQh x SW1 x ... SWk
 psstates - composite process structure states
 psstates = SP1 x SP2 x ... SPn x SL1 x ... SLm

af0 - set of initial allocation functions

TA - set of valid allocation changes

```
TA SUBSET.OF ((AF x psstates x prstates)x(AF x psstates x prstates))
(!A ((A1,x1,y1),(A2,x2,y2)) ELEMENT.OF TA)
[ A1(y1)=x1 AND A2(y2)=x2 AND
  ( (A2 = A1 AND x2 = x1 AND TRS(y1,y2))
  OR (A2 = A1 AND TSS(x1,x2) AND TRS(y1,y2))
  OR (NOT(A1=A2) AND x1=x2) ) ]
```

where TRS = composite state transition rule for PRS

TSS = composite state transition rule for PSS

The process/processor allocation is defined as a triple (AF, af0, TA), where AF is the set of all allocation functions, af0 is a set of initial allocation functions, and TA is a set of transitions between allocation functions. The purpose here is to model the association between processes and processors in the two models. Although most systems maintain a static mapping between processes and processors, this model allows for the specification of systems in which the mappings change over time. The valid sequences of transitions is determined from af0 and TA.

The set of allocation functions AF is defined as the set of all mappings from the set of composite processor states (prstates) onto the set of composite process states (psstates). The set prstates is essentially the cross product of all processor and interconnection states in the processor structure, and the psstates is the cross product of the process and link states in the process structure. The mapping is functional, so that a given state of the processor structure uniquely identifies a state of the process structure. Note that it is possible for an individual process state to have a functional dependency on the states of more than one processor, so that a process can effectively be mapped onto more than one processor or interconnection.

The set of valid allocation changes TA is a subset of ((AF x psstates x prstates) x (AF x psstates x prstates)), where each member of an ordered pair in TA represents the association of an allocation function with a specific processor structure state and a process structure state. Each member of TA represents a change in the system allocation function to be associated with a given change in the processor structure and process structure states. Non-determinism is allowed in that several members of TA may have the same first element but different second elements so that many different changes in the allocation may be possible at a given state of the system. In all cases, the members of TA are subject to the following constraints. First, if the allocation function remains the same across a transition, then a valid change in state of the processor structure must have occurred and the process structure state must have remained the same or undergone a valid transition. Second, if the allocation function changes across a transition then no change in the process structure state must have occurred. These constraints assure that TA is consistent with the transition functions in the processes and processors.

The method by which process/processor allocation is modeled by this structure can be illustrated by two simple examples. The first example is one of modelling a static allocation of processes to processors. Here the allocation is modeled by the allocation function A_0 , with $ALC = (AF, A_0, TA)$. The set TA is then restricted to be a subset of $\{A_0\} \times psstates \times prstates$, subject to the above constraints. As a result, the process/processor allocation reduces to a description of the static mapping between $prstates$ and $psstates$.

The second example is one where the allocation of the system is allowed to change upon the failure of a processor. Assume that the processor structure consists of two processors X and Y with an interconnection I to which they are both connected (see figure E-1). The process structure consists of a single process P, and the initial allocation A_1 has the property that the state of P is dependent solely on the state of processor X. To model the secondary allocation which must occur if X fails, a second allocation function A_2 is specified in which the state of P depends solely on the state of processor Y. Let s be a processor structure state, $XFAIL(s)$ be a predicate asserting that s is a state in which processor X has failed, and $recover(s)$ be a processor structure state which is entered after the recovery process from state s . The required allocation change can be modelled as follows:

```
(!A p) (!A s) [XFAIL(s) IMPLIES ( (A1,p,s),(A2,p,recover(s)) ) MEMBER.OF TA ]
```

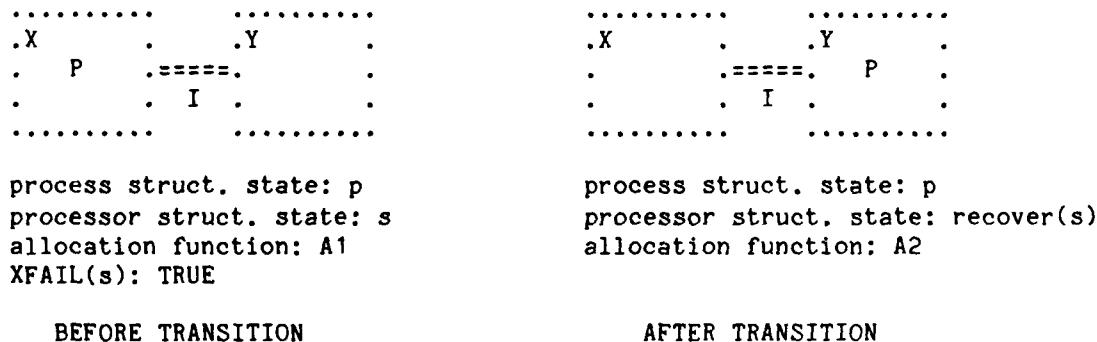


FIGURE E-1.

E.3.2.4 PERFORMANCE SPECIFICATIONS

$PFS = (PSRQ, PRRQ, PSCH, PRCH, PMOD)$

$PSRQ$ - process structure performance requirements
 $PSRQ$ SUBSET.OF ($psstates^*$ x PSA)
 PSA - process structure attributes

$PRRQ$ - processor structure performance requirements
 $PRRQ$ SUBSET.OF ($prstates^*$ x PRA)
 PRA - processor structure attributes

PSCH - process structure performance characteristics
PSCH SUBSET.OF (psstates* x PSA)

PRCH - processor structure performance characteristics
PRCH SUBSET.OF (prstates* x PRA)

PMOD - performance model
PMOD : POWERSET(prstates* x PRA)
--> POWERSET(psstates* x PSA)

PMOD(PPRQ) ==> PSRQ
PMOD(PRCH) ==> PSCH

PSCH ==> PSRQ
PRCH ==> PRRQ

where $\{(x_1, y_1), \dots, (x_n, y_n)\} ==> \{(x_1, z_1), \dots, (x_n, z_n)\}$
IFF $(\forall i) [0 < i < n+1 \text{ IMPLIES } (y_i \text{ IMPLIES } z_i)]$

The performance specification (PFS) is defined as a 5-tuple (PSRQ, PRRQ, PSCH, PRCH, PMOD) where PSRQ is a set of process structure performance requirements, PRRQ is a set of processor structure performance requirements, PSCH is a set of process structure performance characteristics, PRCH is a set of processor structure performance characteristics, and PMOD is a performance model relating the various specifications. The performance requirements specifications serve to establish performance criteria which must be met by the process and processor structures. These requirements are propagated top-down as the performance requirements for a processor structure at one level determine the performance requirements for the process structure at the next level down. The performance characteristics represent derived performance data about the system. These characteristics have a bottom-up dependency as the performance characteristics of the process structure at one level determine the performance characteristics of the processor structure at the next level up.

The performance requirements and characteristics are defined as sets of (state sequence, attribute predicate) pairs. The state sequences are linear sequences of 0 or more composite structure states, and represent various subparts of a process or processor structure functionality. The predicates associate specific attributes with each sequence, such as "elapse time < 100 microseconds". Attributes of the entire structure can be associated with null sequences, such as "system storage < 64K" or "system mass < 40Kg". Using this mechanism, the various quantifiable performance requirements or characteristics can be attached to the process and processor structures at each level.

The performance requirements and characteristics for the process and processor structures must have certain relationships to each other for the design specification to be valid, and these relationships are established through the performance model. The performance model is a function which maps (processor state sequence, attribute) pairs to (process state sequence, attribute) pairs, thus serving to transform the performance requirements/characteristics for a processor structure to a set of

requirements/characteristics for a process structure that it supports. Given the performance model PMOD, then PMOD(PPRQ) must be consistent with PSRQ, and PMOD(PRCH) must be consistent with PSCH, where a requirement R1 is consistent with a requirement R2 iff the attributes associated with sequences in R1 imply the attributes associated with the same sequences in R2. Of course, for the design specification to be valid the performance characteristics in each structure must be consistent with the performance requirements for the structure.

E.3.2.5 BINDING OPTIONS

BD = (FB, FR, BC, eval)

FB - feasible bindings
FR - feasible realizations
BC - binding constraints
eval - binding evaluation function

FB = eval(FR, BC)
NOT(eval(FB, C) = nil)

(PSS U PRS U ALC U PSRQ U PRRQ) SUBSET.OF BC

The binding options (BD) are defined as a 4-tuple (FB, FR, BC, eval) where FB is a set of feasible bindings, FR is a set of feasible system realizations, BC is a set of binding constraints, and eval is a binding evaluation function. The set of feasible realizations FR is a conceptual set of all system realizations which meet the functional requirements of the design specification. The set of binding constraints BC represents a cumulative set of constraints which have been imposed on the system binding through the design process. Constraints which are included in BC are such things as binding restrictions derived from the system requirements, restrictions imposed by the choice of process and processor structures at the "current" and all higher levels in the processing model, and restrictions imposed by the performance requirements. The set of feasible bindings FB is the set of all feasible realizations which are consistent with the binding constraints, i.e. $FB = eval(FR, BC)$. For the system design to be valid, it must also be true that $eval(BD, C)$ is not empty, where C is the set of constraints from the design specification. Hence, there must be at least one binding of the design to a system realization which meets all of the system constraints.

E.3.3 DEFINITION OF THE RELATIONSHIPS BETWEEN SPECIFICATIONS

Although the definitions of the various components of the processing model above define the contents of each part, they provide no definition of the interrelationships between the specifications defined. There are four basic relationships which tie together these specifications, and they are defined below. The first relationship, IMPLEMENTS, is defined in

section E.3.3.1 and defines the requirements that must be met for a design specification of a system to be consistent with the system requirements definition. The SUPPORTS relationship defined in section E.3.3.2 ties a design specification to another design specification which provides the support structure for it. The REFINES relationship presented in section E.3.3.3 describes the relationship between design specifications in the stepwise refinement of a design at one level in the processing model. Finally, the SPECIFIES relationship presented in section E.3.3.4 ties a design specification to a network of physical machines.

E.3.3.1 IMPLEMENTS

```
DS = (PSS, PRS, PFS, ALC, BD=(FB, FR, BC, eval), C)
SRQ = (CM=(ES, ESO, SS, SSO, F), R, CQ, eval)
```

DS IMPLEMENTS SRQ IFF

```
(?E PHI) [ PHI: psstates --> (ES x SS) ]
    where PHI is onto (ES x SS)
    TSS(x1, x2) AND NOT( PHI(x1) = PHI(x2) )
    IMPLIES (PHI(x1), PHI(x2)) MEMBER.OF F
```

```
FR = R
C = CQ
```

The implements relationship is essentially a predicate asserting that a design specification specifies at least one system realization that meets a system requirements specification. The relationship is established if there exists a mapping of composite process structure states in the design specification onto the composite system-environment state set in the system requirements which has the following properties. First, the constraints CQ in SRQ must be the same as the constraints C in DS. Second, the set FR of feasible realizations in DS must be the same as the set R of valid system realizations in SRQ. Finally, any valid transition between states in DS must map onto either no change in states of the conceptual model or a valid change of states of CM. Hence, PHI represents a mapping of the functionality of DS onto the functionality of SRQ, with DS and SRQ required to have the same set of constraints.

E.3.3.2 SUPPORTS

```
DS = (PSS, PRS, ALC, PFS, BD, C, eval)
DS' = (PSS', PRS', ALC', PFS', BD', C', eval)
```

DS' SUPPORTS DS IFF

```
(?E PSI) [PSI : (PS' U LK') --> (PR U IC)
    PSI is onto (PR U IC)
    IF (Pi' MEMBER.OF PS') AND (PSI(Pi') MEMBER.OF LK')
        THEN (!A Lj' MEMBER.OF LK')
            [IF Pi' MEMBER.OF PLj' THEN PSI(Pi') = PSI(Lj')]
    IF (Lj' MEMBER.OF LK') AND (PSI(Lj') MEMBER.OF PR)
```

```

        THEN (!A Pk' MEMBER.OF PLj')
        [(PSI(Pk') = PSI(Lj'))] ]
```

(?E QSI) [QSI : (PR' U IC') --> (PR U IC)
 QSI is onto (PR U IC)
 IF (Qi' MEMBER.OF PR') AND (QSI(Qi') MEMBER.OF IC)
 THEN (!A Wj' MEMBER.OF IC')
 [IF Qi' MEMBER.OF PWj' THEN QSI(Qi') = QSI(Wj')]
 IF (Wj' MEMBER.OF IC') AND (QSI(Wj') MEMBER.OF PR)
 THEN (!A Qk' MEMBER.OF PWj')
 [(QSI(Qk') = QSI(Wj'))]]

(?E PHI) [(PHI : psstates' --> prstates)
 PHI is onto prstates
 IF TSS'(x1', x2') AND NOT(PHI(x1')=PHI(x2'))
 THEN TRS(PHI(x1'), PHI(x2'))]

(!A Qi MEMBER.OF PR) (?E PHI1.i)
 [(PHI1.i : psstates.i' --> SQi)
 PHI1.i is onto SQi
 IF TSS.i'(x1', x2') AND NOT(PHI1.i(x1')=PHI1.i(x2'))
 THEN TQi(PHI1.i(x1'), PHI1.i(x2'))]

(!A Wj MEMBER.OF IC) (?E PHI2.j)
 [(PHI2.j : psstates.j' --> SWj)
 PHI2.j is onto SWj
 IF TSS.j'(x1', x2') AND NOT(PHI2.j(x1')=PHI2.j(x2'))
 THEN TWj(PHI2.j(x1'), PHI2.j(x2'))]

AF' = { A' | A' : prstates' --> psstates' AND
 A' = (Aq1', ..., Aqn', Aw1', ..., Awm') AND
 (!A Qi MEMBER.OF PR)
 [Aqi : prstate.i' --> psstate.i'] AND
 (!A Wj MEMBER.OF IC)
 [Awj : prstate.j' --> psstate.j'] }

where

psstate.i' - composite state of entities mapped by PSI into Qi
 TSS.i' - corresponding state transition rule
 prstate.i' - composite state of entities mapped by QSI into Qi
 TRS.i' - corresponding state transition rule
 and
 psstate.j' - composite state of entities mapped by PSI into Wj
 TSS.j' - corresponding state transition rule
 prstate.j' - composite state of entities mapped by QSI into Wj
 TRS.j' - corresponding state transition rule

(?E KAI) [KAI : (psstate'* x PSA') --> (prstate* x PRA))
 KAI(PSRQ') = PRRQ
 KAI(PSCH') = PRCH]

FB' = eval(FR', BC')

FR' SUBSET.OF FR
BC SUBSET.OF BC'

C' = C

The SUPPORTS relationship between two design specifications DS' and DS states that DS' specifies the support system upon which DS resides, or more specifically that DS' is a design specification for the processor structure of DS. A number of requirements must be met for the relationship to hold, and these are described below. The first set of requirements deals with the mapping of processes, processors, links, and interconnections in DS' to processors and interconnections in DS. For processes and links, there must exist a function PSI from processes and links in DS' onto processors and interconnections in DS. Under PSI, if any process in DS' maps to a link in DS, then all links to which that process is attached in DS' must map to the same link in DS. Similarly, if any link in DS' maps to a processor in DS then all processes connected by the link must also map to the same processor. For processors and interconnections in DS', there must exist a function QSI from processors and interconnections in DS' to processors and interconnections in DS. Under QSI, if any processor in DS' maps to an interconnection in DS, then all interconnections to which that processor is attached in DS' must also map to the same interconnection in DS. Similarly, if an interconnection in DS' maps to a processor in DS then all processors connected by that interconnection must map to the same processor in DS. The result of these restrictions is that the processes, processors, links, and interconnections of DS' are partitioned into disjoint sets according to the processor or interconnection in DS to which they are mapped.

The second group of restrictions deals with the functional mapping between the two design specifications. In a global sense, there must exist a mapping PHI from the composite process structure state (psstates') in DS' to the composite processor structure state (prstates) in DS such that valid transitions in psstates' map to either valid transitions in prstates or no transition. Using the partitioning from above, however, we can qualify the mapping further. For every processor Qi in the processor structure of DS, we can define a composite process state psstates.i' which is the composite state of all links and processes in DS' which map to Qi through PSI. Similarly, for each interconnection Wj in DS we can define a composite process state psstates.j'. We can now define a mapping between the partitions of the processing model in DS' and the processors and interconnections in DS as follows. For each processor Qi in DS, there must exist a mapping PHI1.i from psstates.i' onto SQi such that all valid transitions in psstates.i' map to valid transitions in SQi or no transition in SQi. Also, for each interconnection Wj in DS there must exist a mapping PHI2.j from psstates.j' onto SWj such that all valid transitions in psstates.j' map to valid transitions in SWj or no transition.

Using the partitioning concept a restriction can also be placed on the process/processor allocation in DS'. The requirement is essentially that processes within a partition mapping to a processor Q in DS can only be allocated to processors in DS' which are also mapped to Q. Put more formally, let us define prstates.i' as the composite state of the

processors and interconnections in DS' which map through QSI to processor Qi in DS , and $prstates.j'$ be the composite state of the processors and interconnections which map through QSI to interconnection Wj in DS . Then the allocation functions A' in DS' can be decomposed into subfunctions $Aq1', \dots, Aqn', Aw1', \dots, Awm'$ such that the Aqi map $prstates.i'$ onto $psstates.i'$ and the Awj map $prstates.j'$ onto $psstates.j'$, subject to the state transition consistency rules for allocation functions.

The last set of requirements deals with the constraints and performance requirements/characteristics of the two models. First, there must exist a function KAI which maps $(psstates$ sequences, attribute predicate) pairs onto $(prstates$ sequences, attribute predicate) pairs such that $KAI(PSRQ')$ is consistent with $PRRQ$ and $KAI(PSCH')$ is equal to $PRCH$. Of course, the mapping of $psstates$ ' sequences to $prstates$ sequences implied by KAI must be consistent with the function Φ . In this way KAI provides the translation between the performance requirements and characteristics in DS' and those in DS . Second, the set of constraints C' in DS' must equal the set of constraints C in DS . Finally, FR' must be a subset of FR and SC a subset of BC' . Hence, BC' contains the binding constraints imposed by the design decisions in DS as well as those imposed by design decisions in DS' .

E.3.3.3 REFINES

```

DS    = (PSS, PRS, ALC, PFS, BD, C, eval)
DS'   = (PSS', PRS', ALC', PFS', BD', C', eval)

DS' REFINES DS IFF

(?E PHI) [PHI : psstates' --> psstates
           PHI is onto psstates
           IF TSS'(x1', x2') AND NOT(PHI(x1')=PHI(x2'))
              THEN TSS(PHI(x1'), PHI(x2')) ]

(?E QSI) [QSI : prstates' --> prstates
           QSI is onto prstates
           IF TRS'(y1', y2') AND NOT(QSI(y1')=QSI(y2'))
              THEN TRS(QSI(y1'), QSI(y2')) ]

(!A A MEMBER.OF TA)
  [ x = (x1, ..., xn, ..., xn+m)
    y = (y1, ..., yn, ..., yn+m)
    with    yi MEMBER.OF SQi for 0 < i < n+1
            yi MEMBER.OF SWi for n < i < n+m+1

    A = (A1, ..., An, ..., An+m)
    with    Ai(yi) = xi   for 0 < i < n+m+1  ]

(!A A' MEMBER.OF TA')
  [ x' = (x11', ..., x1k(1)', ..., 
            xn1', ..., xnk(n)', ..., 
            x(n+m)1', ..., x(n+m)k(n+m)') ]

```

```

y' = (y11',...,y1k(1)',....,
      yn1',...,ynk(n)',....,
      y(n+m)1',...,y(n+m)k(n+m)')
with    yij' MEMBER.OF SQij' for 0<i<n+1 and 0<j<k(i)
           yij' MEMBER.OF SWij' for n<i<n+m+1 and 0<j<k(i)

A' = (A11',...,A1k(1)',....,
      An1',...,Ank(n)',....,
      A(n+m)1',...,A(n+m)k(n+m)')
with    Aij'(yij') = xij' for 0<i<n+m+1 and 0<j<k(i)

PHI(..., xi1',...,xik(i)', ...) = (... , xi, ...)

QSI(..., yi1',...,yik(i)', ...) = (... , yi, ... )]

(?E KHI) [ KHI : (psstate'* x PSA') --> (psstate* x PSA)
            KHI(PSRQ') ==> PSRQ
            KHI(PSCH') = PSCH  ]

(?E KSI) [ KSI : (prstate'* x PRA') --> (prstate* x PRA)
            KSI(PPRQ') ==> PRRQ
            KSI(PRCH') = PRCH  ]

FB' = eval(FR', BC')
      FR' SUBSET.OF FR
      BC SUBSET.OF BC'

C' = C

```

During the process of system design it is usually necessary to decompose a design specification at one level in the processing model into a more detailed specification of the same level. It is therefore necessary to define a relationship **REFINES** between a design specification DS and its refinement DS' in order to formalize this notion of decomposition. For the relationship DS' **REFINES** DS to be valid, the following requirements must hold.

The first requirement is that there be a functional mapping between the two design specifications. For the process structures, there must exist a function PHI from the composite process structure state of DS' to that of DS, such that valid transitions in the process structure state of DS' map onto valid process structure state transitions in DS, or no transition. A similar function QSI from the composite processor structure state of DS' to that of DS must also exist.

The second requirement is that the allocation functions of the two design specifications must be consistent with the functional mappings PHI and QSI. Let the allocation functions A in DS be broken up into n+m sub-allocations A1, A2, ...An+m, where n is the number of processors in PR and m is the number of interconnections in IC. If yi is the state of a single processor or interconnection, then xi = Ai(yi) is the composite state of all processes and links which are allocated at least in part to the given processor or interconnection. In the refinement DS', there must be corresponding allocations Aij' and states xij', yij' such that

$A_{ij}'(y_{ij}') = x_{ij}'$, $\text{PHI}(\dots x_{i1}', \dots x_{ik(i)}') \dots = (\dots x_i \dots)$ and $\text{PHI}(\dots y_{i1}', \dots y_{ik(i)}') \dots = (\dots y_i \dots)$. In other words, if a set of processes (or links) ps in DS is allocated to a specific processor (or interconnection) pr in DS , then the set of processes (links) ps' in DS' onto which ps is refined must be allocated within the set pr' of processors (interconnections) which correspond to pr in the refinement. Aside from the restriction to the allocation function, this requires that processors and interconnections be decomposed in a hierarchical manner.

The final set of requirements deals with the consistency of the binding options and performance requirements. For process structures there must exist a mapping KHI from (process state sequence, attribute predicate) pairs in DS' to (process state sequence, attribute predicate) pairs in DS such that $KHI(PSRQ')$ is consistent with $PSRQ$ (see section E.3.2.3), and $KHI(PSCH')$ is equal to $PSCH$. For processor structures, there must exist a similar mapping KSI between (processor state sequence, attribute predicate) pairs in DS' and DS such that $KSI(PPRQ')$ is consistent with $PPRQ$ and $KSI(PRCH')$ is equal to $PRCH$. Next, the set of constraints C' in DS' must equal the set of constraints C in DS . Finally, the set of feasible realizations FR' in DS' must be a subset of FR , and BC' a superset of BC .

E.3.3.4 SPECIFIES

DS = (PSS, PRS, ALC, PFS, BD, C, eval)
MCH = (PSS", PRS", ALC", PFS", BD", C", eval)

DS SPECIFIES MCH IFF

MCH SUPPORTS DS

QSI is one-to-one

SOFTWARE(PSS")
HARDWARE(PRS")
STATIC(ALC")
i.e., $TA" = ((\{A"\} \times psstates" \times prstates")^{**2})$

A processing model is in a sense complete when its lowest level design specification can be mapped onto a physical set of hardware and software. Given a machine level specification MCH and a design specification DS , we say that DS SPECIFIES MCH if the following requirements are met. First, the machine level specification MCH must have its processor structure map directly to a hardware implementation and its process structure map directly to a software implementation on the given hardware. Second, the process/processor allocation in MCH must be static (see section E.3.2.3). Third, the relationship MCH SUPPORTS DS must be valid. Finally, the function QSI mapping processors and interconnections in MCH to processors and interconnections in DS must be one to one.

E.4 HARDWARE/SOFTWARE REQUIREMENTS DEFINITION

HSRQ = (SFRQ1, ..., SFRQn, HDRQ)

SFRQi - subsystem software requirements
HDRQ - hardware requirements

SFRQ = (DS, SINT, SBIND, SOFT)

DS - subsystem design specification
SINT - interface specifications
SBIND - software subsystem binding function
SBIND : PSS --> SOFT
SBIND is onto SOFT
SOFT - existing software

HDRQ = (DS, HINT, HBIND, HARD)

DS - subsystem design specification
HINT - interface specifications
HBIND - hardware subsystem binding function
HBIND : PRS --> HARD
HBIND is one-to-one
HARD - existing hardware

PM BOUND.TO HSRQ IFF

PM = (DS1, ..., DS_n)

PM SBOUND.TO MCH

HSRQ = (SFRQ1, ..., SFRQ_{n+1}, HDRQ)

SFRQi = (DS_i, SINT_i, SBIND_i, SOFT_i) 0 < i < n+1
SFRQ_{n+1} = (MCH, SINT_{n+1}, SBIND_{n+1}, SOFT_{n+1})
HDRQ = (MCH, HINT, HBIND, HARD)

The hardware/software requirements comprises the total output of the system design stage. In essence it is a list of software requirements specifications along with a hardware requirements specification, where these requirements are defined below. The hardware/software requirements are of course derived from the processing model constructed during the system architecture design phase, and so the binding requirements between the processing model and the hardware/software requirements are discussed below.

The software requirements component of the hardware/software requirements is defined as a 4-tuple (DS, SINT, SBIND, SOFT), where DS is a design specification, SINT is a set of interface specifications, SBIND is a software subsystem binding function, and SOFT is a set of existing software components. The design specification DS is that portion of the processing model from which this particular software requirements specification is derived, and it is the source of the functional requirements, performance requirements, and constraints which are to guide the software design process. The interface specifications are a formal specification of the interfaces between different software components at

this level and the interface to the subsystem at the next higher level in the processing model. SBIND is a function which maps PSS (from DS) onto SOFT, where SOFT is a set of software components used in the implementation of the subsystem.

The hardware requirement specification is defined as a 4-tuple (DS, HINT, HBIND, HARD), where DS is a design specification, HINT is a set of hardware interfaces, HBIND is a hardware binding function, and HARD is a set of hardware components. Again, DS provides the functional requirements, performance requirements, and constraints to drive the hardware design. HINT specifies the interfaces between the different hardware components in the design, as well as the interface between the hardware and the software which executes on it. Finally, HBIND is a function which maps PRS (from DS) onto HARD, where HARD is a set of hardware components used in the implementation of the subsystem.

The purpose of the system binding phase is to create the HSRQ so that the relationship PM BOUND.TO HSRQ is valid. Given a processing model $PM = (DS_1, DS_2, \dots, DS_n)$ and a hardware/software requirements specification $HSRQ = (SFRQ_1, \dots, SFRQ_{n+1}, HDRQ)$, then PM BOUND.TO HSRQ is valid if and only if the following relationships hold. First, PM must be superficially bound to a net of machines MCH (see section E.3.1). Second, each set of software requirements in HSRQ must be derived from their corresponding design specifications in PM, with an extra set of software requirements derived from the specification MCH. Put more directly, $SFRQ_i = (DS_i, SINT_i, SBIND_i, SOFT_i)$ for all DS_i in PM, $SFRQ_{n+1} = (MCH, SINT_{n+1}, SBIND_{n+1}, SOFT_{n+1})$, and $HDRQ = (MCH, HINT, HBIND, HARD)$. These two requirements complete the linkage between the processing model and the hardware/software requirements.

E.5 CONCLUSIONS

The models presented above serve as a conceptual model for the specification languages needed in the system design stage. As such, they specify what concerns need to be addressed by these languages but not how they are addressed. It is not intended that any actual specification language directly specify the components of these definitions, as certain sets such as the set of legal state transitions in the conceptual model would defy total enumeration. It is necessary, however, for any specification language used to be conceptually mappable into the models provided here (thus defining the semantics of that specification language).

Although the structure of the models given does contain some implications concerning the system design methodology, it is not a specification for any particular methodology (as a conceptual model for a system is not a specification for a single implementation of that system). For instance, although the processing model is structured in a top-down hierarchical manner, since it is merely a model of the interface between the architecture design phase and the binding phase it does not preclude the use of a bottom-up oriented design methodology within the system architecture design phase. Neither is it necessary that the processing model be completed before binding is undertaken. What these models do provide is a formal definition of the specifications which are the core of the system design process, and so provide the means by which the formal specification of a system design methodology can be constructed.

APPENDIX F

A RIGOROUS APPROACH TO BUILDING FORMAL SYSTEM REQUIREMENTS

INTRODUCTION

Formal definition of system requirements has received considerable attention in recent years. A measure of the researchers' concern with this topic is the great variety of specification language proposals that have been put forth. They differ in the degree of formality, power, and the nature of the formalisms being used. Some approaches are based on the use of finite-state machines [HENI79] and, thus, they offer simplicity but also reduced power. Others emphasize dataflow (SADT [ROSS77], PSL/PSA [TEIC77]). They are concerned with the functions to be performed by the system and the data being passed between them. Yet another approach is used in RSL [BELL77] which defines the requirements in terms of stimulus-response paths. Attempts have also been made to capture the behavior characteristics of the systems in algebraic specifications, e.g., [RIDD78], and as partial orders [GREI77]. Data-oriented modeling of the requirements has been stimulated by efforts in the database area [SMIT79] while applicative languages have been advocated as a means to achieve executability of the requirements [ZAVE81].

The dominant concerns of those involved in the development of requirements specification languages have been the ease of use and the potential for automation of their respective proposals. There are, however, a number of other important issues demanding careful investigation. They relate to the pragmatics of using formal specifications. How one chooses an appropriate specification language, how one develops the specifications, how one gets started, are questions often formulated by designers that feel the need for improvements in the quality of the system requirements but have no experience with the use of formal specifications. These questions also explain why formal specifications are rarely used in practice despite the great need to accumulate experience in this area and despite the benefits they promise.

This paper addresses several of these issues. It reports on the author's experience in the use of formal specifications and presents a step by step approach to developing formal requirements. The tutorial is intended to give assistance and confidence to the novice and to share with other practitioners some observations about the nature of system requirements. A basic knowledge of predicate calculus and set theory is assumed on the part of the reader. While no exposure to any requirements definition language is required, some appreciation for the role the requirements play in the development of the system is necessary.

The remainder of the presentation is separated into four sections. The first one introduces a formal model of system requirements. A systematic approach to developing formal requirements by starting with the general model and by adapting it to the needs of the problem at hand is described and illustrated by means of a simple but realistic example in the section that follows. A discussion of several topics related to the development of formal requirements, including unresolved issues and current concerns, precedes the conclusions.

FORMAL MODEL OF SYSTEM REQUIREMENTS

The system requirements are generated in the problem definition stage and prior to any attempt at system design. They consist of a conceptual model and a set of constraints which together define the acceptability criterion for any proposed system realization:

$$SRQ = (CM, CQ)$$

CM - conceptual model
CQ - set of design constraints (implicit/explicit)

(See the notation summary for conventions used in this paper.) A system is said to meet its requirements if and only if it carries out the functionality described by the conceptual model and satisfies all relevant constraints.

The role of the conceptual model is to capture in finite and precise terms the nature of the interaction between the needed system and its environment. The constraints, on the other hand, limit the design space by imposing restrictions over the class of systems the designer might consider. Actually, the degree of complexity of a system is measured not by size alone but also by the severity of the constraints it must satisfy.

Before continuing the discussion of the conceptual model which is the main concern of this paper, it ought to be pointed out that recent increases in the ability to formally define the desired functionality have not been accompanied by commensurate advances in the definition of system constraints. There are four important reasons contributing to this state of affairs. First, there is a great diversity of types of constraints (e.g., response time, space, reliability, cost, schedule, weight, power, etc.). Second, some of them are related to possible design solutions which are not formally stated at the time the system requirements are conceived. Furthermore, their relevance differs at different points in the design. Third, many constraints (e.g., maintainability) are not formalizable given current state-of-the-art. Finally, not all constraints are explicit. For instance, the designer is expected to follow generally accepted rules of the trade in designing a system without having them explicitly stated.

In general, there is considerable agreement among authors with regard to the nature of the conceptual model. The conceptual model must have the ability to describe all pertinent environmental states, an abstraction of the system states, and the way in which both environmental and system states change.

$$CM = (E, E_0, S, S_0, F)$$

E - environmental states
E₀ - possible initial environmental states
S - system states
S₀ - possible initial system states
F - state transition rules
F SUBSET.OF ((E x S) x (E x S))

The definition above makes clear two important facts. First, because both the environmental and the system states may be infinite in number, the model may not be reduced, in general, to a finite-state machine. Second, in the general case, the state transition rules define a relation between pairs of states because nondeterminism is present in most systems and their environments.

The approach to describing the states and the state transition rules varies from one specification language to another. The notation used in the next section, for instance, is borrowed from set theory (for describing the environmental and the system states) and predicate calculus (for defining the state transition rules). Furthermore, some languages make implicit assumptions about either or both the nature of the states and of the state transition rules; the loss in generality is motivated by increased specificity in the handling of a particular application area. As an example, given a system that responds to stimuli from the environment in a manner which is independent of the history of previous stimuli and responses, it may be easily described in a language which equates the state of the environment with the current stimulus, which has no ability to describe system states, and which is able to define a mapping from the set of stimuli to the set of responses. Yet another example could be used to illustrate the fact that there is also great variability in the way state transitions may be described: in a biomedical simulation system a new state is generated as a result of the integration of a set of differential equations.

If the conceptual model is structured in a hierarchical manner, e.g., $CM'' = (CM, CM')$, then one needs to have defined the notion of decomposition as shown below. CM' is said to be a decomposition of CM , i.e., CM' REFINES CM , if and only if

given $CM = (E, E_0, S, S_0, F)$ and $CM' = (E', E'_0, S', S'_0, F')$

there exists a function PHI such that

- a. $\text{PHI} : (E' \times S') \rightarrow (E \times S)$
- b. PHI is onto $(E \times S)$
- c. for-all e'_0, s'_0 there-exist e_0, s_0 : $\text{PHI}(e'_0, s'_0) = (e_0, s_0)$
- d. IF $((e'_1, s'_1), (e'_2, s'_2)) \in F'$ AND
 $\text{PHI}(e'_1, s'_1) = (e_1, s_1)$ AND
 $\text{PHI}(e'_2, s'_2) = (e_2, s_2)$ AND
 $\text{NOT}((e_1, s_1) = (e_2, s_2))$
THEN $((e_1, s_1), (e_2, s_2)) \in F$

In the case of large systems, where top-down specification of the conceptual model becomes a necessity, this definition establishes a fundamental criterion for checking the self-consistency of the system requirements.

THE APPROACH

This section introduces the reader to a systematic approach to developing formal system requirements. The formal definition of system requirements and an informal description of the application are the starting point for this approach. The formal definition of system requirements was given in the previous section. The application considered here is a simplified version of a banking operation:

EGBANK is a small bank having several branch offices in the city. Tellers from each branch office are authorized to create new accounts, to check the balance of some account, to make deposits and withdrawals from one account at a time, and to transfer money from one account to another. All successful banking transactions are logged for auditing purposes. The log entries always include the teller identification.

The way in which the informal specification is converted in a formal conceptual model is outlined below.

Preliminary tailoring of the formal model.

Most applications do not require the full power of the formal requirements definition model. This explains why in some cases even finite-state machines proved adequate [HENI79]. Early identification of the complexity of the state transition rules may bring about significant savings in the effort involved in generating the requirements. This statement is strongly supported by past experience and may be explained by the fact that, by understanding the exact nature of the transition rules one is better prepared to avoid two opposite but equally time wasting pitfalls: (1) the use of formalisms which are not powerful enough to do the job and (2) the generation of specifications which are unnecessarily complex and whose simplification often turns out to be more expensive than starting from scratch.

Fortunately, a priori determination of the nature of the state transition rule appears to be possible. By analyzing the informal problem definition one may be able to establish if the state transition rule, F , is indeed a relation or maybe a function. In general, nondeterministic behavior suggests the use of a relation rather than a function, i.e., given the current system/environment state there are several possible next states.

In EGBANK, the state of the environment is given by the nature of the currently pending teller queries. The state of the system is represented by the composite of all bank accounts. State changes in the environment occur due to arrival of new customers which trigger new queries in their behalf and due to arrival of replies to pending queries. State changes in the system take place due to processing of queries which may change the amounts present in various accounts and may create new accounts.

It appears, therefore, that F needs to be defined as the relation

F SUBSET.OF ((E x S) x (E x S))

where both E and S are non-finite. While this degree of complexity seems unavoidable, there are still some opportunities for simplifications and they should be investigated. For instance, F may be decomposable into simpler relations or functions.

The suggestion has been made earlier that the state of the environment is determined by the pending queries. An argument could be made, however, that the system is affected not by the pending queries, but by the processing of each new query. Furthermore, the answer to a query is determined by the nature of the respective query and by the state of the system at the time the query is processed (not at arrival time). Consequently, one component of F could be

FS : (Q x S) --> (R x S)

where

Q is the set of possible queries
R is the set of possible replies
S is the set of system states (as before).

Each query may be considered as if it were alone in the system because this is exactly the tellers' perception of the system.

As far as the environment is concerned, one may define a relation FE which captures the changes that occur at each teller: the issuing of some query from Q, the return of some answer from R, and the lack of activity which is denoted by "nil".

FE SUBSET.OF (E x E)

where

n is the number of tellers

E = (Q UNION R UNION {nil})**n

```
for-all x,y:  
  [ ((...,x,...), (... ,y,...)) MEMBER.OF FE  
    IFF  
    ( (x MEMBER.OF Q) AND (y MEMBER.OF R) AND id(x)=id(y) )  
    OR ( (x MEMBER.OF R) AND (y = nil) )  
    OR ( (x = nil) AND (y MEMBER.OF Q) ) ]
```

The function "id" appearing above will be formally defined later. It was introduced here, however, in order to state that the answer (i.e., y) received by some teller bears the same identification as the original query (i.e., x) sent by the same teller.

At last F may be defined by using FS and FE:

```
( (e1, s1), (e2, s2) ) MEMBER.OF F
IFF
(e1, e2) MEMBER.OF FE
AND
IF ( ( e1=(...,x,...) AND e2=(...,y,...) AND
      (x MEMBER.OF Q) AND (y MEMBER.OF R) )
      THEN ( ((x, s1), (y, s2)) MEMBER.OF FS )
```

While it is true that F could have been defined directly in terms of S, Q, R, and n, there are certain advantages to the strategy being presented. In many cases, the attempt to look for a decomposition of F results in much simplified formalizations. More importantly, however, it often leads to a degree of separation of concerns useful in the understanding and analysis of the requirements. In our example, for instance, FS deals with the query processing aspect while FE relates to behavioral aspects of the environment indicating such things as the fact that a reply must succeed the respective query, that queries are not necessarily processed in the order of arrival, etc.

Definition of the environmental states.

The informal requirements specify the nature of the queries (Q) and, indirectly, the nature of the replies (R) that have been used in the high level definition of the environmental states. There are four types of queries: account creation, reading of the account data, updating of the account, and fund transfer between two accounts. Furthermore, each account is generally characterized by the owner's name, by the amount on deposit, and by some account number. By taking advantage of this knowledge and by the fact that each query must have a teller identifier, the set Q may be now defined

```
Q = tellerids x
    ( ({create} x accounts x customers x deposits)
     UNION
     ({read} x accounts)
     UNION
     ({update} x accounts x deposits)
     UNION
     ({trans} x accounts x accounts x deposits) )
```

The sets tellerids, accounts, customers, and deposits are left undefined in this paper because their definitions are trivial to construct, not from the earlier informal specification of the problem but by soliciting the missing information. This illustrates one important advantage of the formal specifications. They frequently uncover many important details that were left out in the informal requirements definition.

If one assumes that all replies must be complete, i.e., they must include the account number, customer name, and current deposit value, then R is defined as follows:

```
R = tellerids x
  ( {error}
    UNION
    (accounts x customers x deposits)
    UNION
    ( (accounts x customers x deposits)
      x
      (accounts x customers x deposits) ) )
```

At this point, it is easy to see how the id function introduced earlier works. It simply returns the first element of any n-tuple supplied as its argument.

Definition of system states.

The state of the system is characterized, at any point in time, by the status of all the accounts and by the transaction log. Therefore, the system state might have to be defined in terms of a set that captures the state of the accounts (call it "b" from bank records) and by a sequence that models the log (call it "l").

```
s = (b, l)
```

Next, one could propose b to be described by

```
b SUBSET.OF (accounts x (customers UNION {nil}) x deposits)
```

This definition, however, would permit the undesirable situation where the same account appears twice in the system. Both (1234, Smith, \$350) and (1234, Brown, \$250) could be part of S. The single account number occurrence condition forces b to be a function from accounts to customers cross deposits:

```
b : accounts --> (customers x deposits)
```

Notation wise, however, later use of b will be permitted to take both the form of a set (e.g., (a,c,d) MEMBER.OF b) and that of a function (e.g., b(a)), depending of which one is more convenient at the time.

Given the nature of the log which is only a sequence of queries, i.e.,

```
l MEMBER.OF Q*,
```

the set of system states, S, becomes

```
S SUBSET.OF {b | b : accounts --> (customers x deposits)} x Q*
```

This completes the definition of the system states.

Definition of state transition rules.

Because part of the definition was already given earlier in this section, all that remains to be done is to complete the definition of FS which has been established to be of the form:

FS : (Q x S) \rightarrow (R x S)

For the sake of clarity, separate definitions are provided for each of the four types of queries.

The creation of a new account requires that account to be unassigned to any customer. Furthermore, at creation time, both the customer name and some value for the initial deposit must be provided. The account number is supplied by the teller.

```
FS((id,create,a,c,d), (b,l))
  <= if  (a MEMBER.OF accounts)  AND
        (c MEMBER.OF customers) AND
        (d MEMBER.OF deposits)  AND
        ( (a,nil,0) MEMBER.OF b )
    then
      ((id,a,c,d),
       ((b MINUS {(a,nil,0)} UNION {(a,c,d)}),
        ((id,create,a,c,d).l)))
    else
      ((id,error), (b,l))
```

It should be noted that in case the query is improperly specified, the reply being returned is an error message. Moreover, errors are not logged.

In order to find out information about an account, its number has to be supplied.

```
FS((id,read,a), (b,l))
  <= if  (a MEMBER.OF accounts)  AND
        (there-exist c,d: ((a,c,d) MEMBER.OF b ))
    then
      ((id,a,c,d), (b,((id,read,a).l)))
    else
      ((id,error), (b,l))
```

While testing the fact that "a" is a valid account is mathematically redundant, it is kept in the definition for clarity purposes. (This issue often causes long discussions during requirements reviews but it is the author's strong conviction that clarity must come before brevity.)

Both deposits and withdrawals are accomplished via the update query. It depends upon the sign of the amount being supplied by the teller. An additional condition for the success of the query is that the amount left in the account must be strictly positive.

```
FS((id,update,a,d), (b,1))
<= if (a MEMBER.OF accounts) AND
      (d MEMBER.OF deposits) AND
      (there-exist c,d0: ( (a,c,d0) MEMBER.OF b ) AND
       (d0+d > 0) )
      then
        ((id,a,c,d0+d),
         ((b MINUS {(a,c,d0)} UNION {(a,c,d0+d)}),
          ((id,update,a,d).1)))
      else
        ((id,error), (b,1))
```

The transfer query, called trans, allows one to transfer funds between two accounts. Its meaning is analog to that of removing a positive amount from the first account followed by a deposit in the second account.

```
FS((id,trans,a1,a2,d), (b,1))
<= if (a1 MEMBER.OF accounts) AND
      (a2 MEMBER.OF accounts) AND
      (d MEMBER.OF deposits) AND (d>0) AND
      (there-exist c1,d1: ( (a1,c1,d1) MEMBER.OF b )
       AND (d1-d > 0) ) AND
      (there-exist c2,d2: ( (a2,c2,d2) MEMBER.OF b ) )
      then
        ((id,a1,c1,d1-d,a2,c2,d2+d),
         ((b MINUS {(a1,c1,d1)}) UNION {(a1,c1,d1-d)}
          MINUS {(a2,c2,d2)}) UNION {(a2,c2,d2+d)}),
          ((id,trans,a1,a2,d).1)))
      else
        ((id,error), (b,1))
```

The entire specification is complete. Its accuracy still needs to be established through reviews involving the intended user or customer.

Because increases in the complexity of the items being described results in more complex definitions, the introduction of more powerful notation than the one employed in this paper becomes a necessity. At a minimum, one needs to formulate the definitions in terms of primitive functions which are separately defined at some later point. Ultimately, the designer is led naturally, by the need for clarity and simplicity, to developing hierarchical specifications.

DISCUSSION

The approach described in this paper has grown out of the experience gained in the last four years during which formal requirements have been defined for a large variety of relatively small problems. They included the semantic definition of numerous toy languages, the specification, at several levels, of the message handling subsystem for a local communication network, the definition of the communication primitives for a proposed message based version of Pascal, and some data processing applications comparable to the example used in the previous section. Involvement in the development of requirements for some real production systems using a partially-formalized technique described in [ROMA79] also helped in better understanding what is pragmatically feasible with regard to the production use of formal requirements.

It is the contention of this paper that the approach is ready for use in small to medium size data processing and real time systems. Nevertheless, special consideration must be given to the nature of the problem being considered, the background of the personnel involved, the formalism being contemplated, and to the balance between the formal and the informal components of the requirements to be produced.

The introduction of formal requirements into an organization can be neither sudden nor complete. A wise first step is to employ formal requirements only for the hard-to-define aspects of the system requirements in conjunction with some other less formal but already familiar technique. Thus, the few designers who happen to have the appropriate formal background may be utilized effectively and the initial learning curve has a minimal effect on the overall project performance.

Furthermore, the combined use of formal and informal specifications appears to be not just a transient solution meant to bring about the widespread use of formal specifications but a highly desirable property of requirements definitions in general. Experience has shown that, even when the requirements are completely formalized and the people involved have above average mathematical skills, the absence of an accompanying informal narrative greatly increases the review time and reduces their effectiveness. Finally, one other important factor affecting the success of a formal specification is the use of standard notation. Future use of computer-aided design tools will make this issue obsolete but, until then, lack a standardization may result in misunderstandings and confusion. (Our policy has been to stay with basic mathematical notation. However, the use of a standard keyboard character set has resulted in some compromises.)

One issue that has been ignored throughout most of this paper is the formal definition of the system constraints. So far, the formal requirements we developed centered on rendering the system functionality (the conceptual model) and allowed the constraints to be formulated through the use of natural language. However, attempts to formally specify some of the constraints have been made by others, e.g., [ALFO79, ZAVE81]. Our own preliminary results indicate that different approaches are required in order to deal with different types of constraints. Compulsory distribution of some of the data or processing, for instance,

may be defined by a site function which maps entities of the conceptual model (e.g., events or state transitions) into a set of locations. Other constraints, such as response time, involve a mapping from sequences of events to some appropriate set of values. A well-defined approach, however, still needs to be developed and evaluated.

More work is also required to establish techniques for checking the self-consistency, completeness and accuracy of the requirements. Ad-hoc mathematical proofs and group reviews proved adequate for small scale problems but are not expected to be cost effective and accurate enough for large projects where the use of computer-aided design tools able to carry out some of these checks and proofs becomes a necessity.

CONCLUSIONS

A rigorous approach to the development of formal system requirements definitions has been presented in a tutorial fashion and illustrated on a simple example of a banking system. The approach reflects the author's several years experience with developing formal requirements for a variety of small scale problems. The notation used in the paper is based on set theory and predicate calculus both of which are generally considered essential in the education of the today's computer scientist and are familiar to many system designers. The paper contends that, based on the experience accumulated with the use of both formal and semi-formal specifications, the development of formal requirements for small to medium size systems is feasible and can be cost effective.

REFERENCES

- [ALFO79] Alford, M., "Requirements for Distributed Data Processing Design," Proc. 1st Int. Conf. on Distributed Computing Systems, pp. 1-14, October 1979.
- [BELL77] Bell, T. E., Bixler, D. C. and Dyer, M. E., "An Extendable Approach to Computer-Aided Software Requirements Engineering," IEEE Trans. on Soft. Eng. SE-3, No. 1, pp. 49-60, January 1977.
- [GREI77] Greif, I., "A Language for Formal Problem Specification," CACM 20, No. 12, pp. 931-935, December 1977.
- [HENI79] Heninger, K. L. "Specifying Software Requirements for Complex Systems: New Techniques and Their Applications," Proc. Conf. on Specifications of Reliable Software, April 1979.
- [RIDD78] Riddle, W. E., et al, "Behavior Modeling During Software Design," IEEE Trans. on Soft. Eng. SE-4, No. 4, pp. 671-678, July 1978.
- [ROMA79] Roman, G.-C., "Verification Procedures Supporting Software Systems Development," Proc. of 1979 NCC, pp. 947-956, June 1979.
- [ROSS77] Ross, D. T., "Structured Analysis (SA): A Language for Communicating Ideas," IEEE Trans. on Soft. Eng. SE-3, No. 1, pp. 16-34, January 1977.
- [SMIT79] Smith, C. and Browne, J. C., "Modeling Software Systems for Performance Predictions," Proc. Computer Measurement Group X, pp. 321-341, December 1979.
- [TEIC77] Teichroew, D. and Hershey, III, E. A., "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems," IEEE Trans. on Soft. Eng. SE-3, No. 1, pp. 41-48, January 1977.
- [ZAVE81] Zave, P. and Yeh, R. T., "Executable Requirements for Embedded Systems," Proc. 5th Int. Conf. on Soft. Eng., pp. 295-304, March 1981.

NOTATION SUMMARY

n-tuple	(a, b, ...)
set definition	{a, b, ...} {x predicate(x)}
function definition	fname: domain --> range fname(arguments) <-- if predicate then value1 else value2
quantifiers	
existential	(there-exist x: predicate)
universal	(for-all x: predicate)
cross product	set1 x set2
n'power cross product	set##n
logical implication	IF predicate1 THEN predicate2
logical operators	AND, OR, NOT
subset test	set1 SUBSET.OF set2
set membership test	element MEMBER.OF set
set operations	UNION, INTERSECTION, MINUS
set of all sequences	
over some set	set*
concatenation	sequence1.sequence2
special constant	nil

APPENDIX G
FUNCTIONAL SPECIFICATION OF DISTRIBUTED SYSTEMS

INTRODUCTION

The potential for a major qualitative improvement in the effectiveness of systems development rests to a large extent on the availability of appropriate specification languages. While establishing the basis for precise communication, formal specifications also open the doors to extensive systematic (mental or automated) system design analysis techniques whose scope would ultimately include logical verification, performance checking, automatic generation of predictive models, and more. Such advances in system design technology are presumed to pave the way for powerful development tools which are very much needed at a time when system development productivity registers relatively minor increases and personnel costs are on the rise.

Specification languages have received considerable attention both in industrial and academic circles. Various proposals range in flavor from tables, standardized document formats, and graphic representations [ROSS77], at one extreme, to formal languages having well-defined syntax and semantics [ROBI77] at the other. The work on program specifications has largely dominated the field, both with respect to the attention received and level of formality. (The reader is referred to [LISK79] for a good survey of available formal program specification techniques.) This is in part due to the strong influence exercised by related research in the programming language area (CLU [LISK77], Alphard [WULF76], etc.).

Despite the considerable effort that has been expended in recent years in the study of parallel computation and distributed systems design, the specification of distributed systems continues to present designers with many unresolved problems [LISK79]. Some programming and program specification languages (Path-Pascal [CAMP79], DREAM [RIDD78], etc.), while able to express concurrency, are limited in their capacity to deal with distribution and restrict the designer's freedom to define arbitrary communication protocols. Furthermore, less formal approaches (e.g., RSL [BELL77]), while valuable from a pragmatical viewpoint, are only a temporary solution that sets the stage for future assimilation of theoretical results into the production environment. This paper reports on one effort to develop a formal Distributed Systems Design Language (DSDL) and the conclusions reached after experimentation with the language on several case studies. The hope is for this work to provide valuable insights that could affect the next generation of distributed systems specification languages.

In DSDL, systems are described as nets of communicating processes. Each process in the net has its own local data over which it has sole control, procedures that specify primitive and indivisible operations over the data, and possesses the ability to exchange messages with other processes in the net. The behavior of the process specifies the order in which its procedures are invoked. Sequences of procedure invocations, also called event sequences, are allowed to execute concurrently within the process.

A net is defined by its processes, by the logical communication links, and by the communication protocols associated with the individual links. Among the processes of a net, some are used to model its

environment; they are called external processes. The links identify the logical connections between processes. Several processes may be associated with the same link and the same process may use several links. The way in which an individual link behaves is stipulated by the communication protocol associated with the respective link.

Several considerations have influenced heavily the nature of the DSDL: the emphasis on formality, the desire to promote the principle of separation of concerns, the need to support hierarchical specifications, and the aim toward generality. Formality is achieved through the use of set theoretical models for data representation, by employing predicate calculus in defining the procedures (using input/output assertions), etc. The principle of separation of concerns is reflected by the manner in which the definitions of the net and of the process are structured; they are meant to enhance the designer's ability to describe the system in terms of clean abstractions. Hierarchical descriptions of the system are enabled by the fact that processes may be refined into nets. Finally, the generality of the language is enhanced by its capacity to describe a variety of communication structures and protocols.

The language, as described here, is concerned only with the functional specification of distributed systems. However, the addition of performance specifications to DSDL is currently under investigation and is anticipated to share the direction adopted in [BOOT80, SMIT79, SANG79]. The ability to relate in a direct and simple fashion functional and performance aspects of distributed systems is expected to contribute to the enhancement of the designer's ability to choose objectively between alternate solutions based on performance analysis of the various candidates.

The next section introduces DSDL by means of a highly simplified annotated example representative of the nature of the language. While many of the language features may still remain rather obscure after scanning the example, the subsequent section refers back to the example as an illustration for the definition of the language syntax and semantics, thus removing any ambiguities one might read into the example. The language definition is followed by a review of several open issues and preliminary research results.

LANGUAGE ILLUSTRATION

The purpose of this section is to introduce the reader to the various DSDL features by means of a simple annotated example. It is intended to provide a concrete reference point for the formal definitions of process and net described in the next section. A highly simplified version of a distributed banking system forms the basis for this illustration. Upper case words indicate language defined entities while lower case words denote terms selected by the designer.

```
NET ebank. /* EGBANK is a small bank having two branch offices in the city.  
/* Tellers from each branch office are authorized to create new  
/* accounts, to check the balance of one or more accounts, to  
/* make deposits and withdrawals from one account at a time, and  
/* to transfer money from one account to another. These activities  
/* are supported by a computer system consisting of three  
/* components that communicate with each other via messages.  
/* Each branch office interacts with a local process which, in  
/* turn, has access to a database located elsewhere.
```

```
DEFINE office: PROCESS. /* Definition of a class of processes.
```

PARAMETERS.

```
CONST id: INTEGER; /* Branch office identifier.  
CONST db: PROCESS; /* Process controlling the databank.  
CONST tty: EXTERNAL PROCESS; /* Local data entry sources.  
CONST ttylink: LINK; /* Connection to data entry sources.
```

DATA.

```
VAR /* Request counter and message identifier.  
n: INTEGER; n>0;  
CONST /* Terminal identifiers.  
Terminals={z | 0<z<21 AND INTEGER(z)};  
CONST /* Definition of acceptable input commands, i.e., requests.  
Req ={('create',c),('update',a,v),('transfer',a1,a2,v),  
('read',a[1],...,a[m]) | INTEGER(m) AND m>0 };
```

INITIALIZATION.

```
n = 1;
```

```
PROCEDURE w:=format(z). /* Message is formed for transmittal to db.  
IN: z MEMBER.OF (Terminals X Req) AND z=(trm,rq);  
OUT: n'=n+1 AND w'=(id,n,trm,rq);  
EXPT: w'=NIL;
```

```
PROCEDURE w:=reply(z). /* Reply from db is prepared for the teller.  
IN: z=(id,no,trm,ans) AND no< n AND trm MEMBER.OF Terminals;  
OUT: w'=(trm,ans);  
EXPT: w'=NIL;
```

```
BEHAVIOR.
PARBEGIN
    BEGIN /* Terminal requests are accepted, formated, and sent to the
           /* database for processing. Invalid queries are rejected.
        LOOP {GET(z) FROM tty ON ttylink; w:=format(z);
              IF NOT.NIL(w) THEN SEND(w) TO db ON switch;
              ELSE SEND(w) TO tty ON ttylink;}
    END.

    BEGIN /* Replies from the db are sent to the terminals.
        LOOP {GET(z) FROM db ON switch; w:=reply(z);
              IF NOT.NIL(w) THEN SEND(w) TO tty ON ttylink;}
    END.
PAREND.
END-DEFINE office.
```

```
EXTERNAL PROCESS tty1: UNDEFINED;
EXTERNAL PROCESS tty2: UNDEFINED;
```

```
PROCESS branch1: office. /* Local processing for branch1.
PARAMETERS.
    id      = 1;
    db      = database;
    tty     = tty1;
    ttylink= ttylink1;
END-PROCESS branch1.
```

```
PROCESS branch2: office. /* Local processing for branch2.
PARAMETERS.
    id      = 2;
    db      = database;
    tty     = tty2;
    ttylink= ttylink2;
END-PROCESS branch2.
```

```

PROCESS database.      /* Central databank.

DATA.
  VAR /* Input buffer for messages received from the branch offices.
        ibuffer ={z | z SUBSET.OF (? X ? X ? X Req)};
  VAR /* Output buffer for messages to be sent to the branch offices.
        obuffer ={z | z SUBSET.OF (? X ? X ? X Ans)};
CONST /* Definition of acceptable requests.
  Req      ={('create',c),('update',a,v),('transfer',a1,a2,v),
            ('read',a[1],...,a[m]) | INTEGER(m) AND m>0 }
CONST /* Definition of database replies.
  Ans      ={'error'} UNION (Accounts X Names X INTEGER)*;
VAR /* Databank containing account information.
  Records SUBSET.OF (Accounts X Names X INTEGER);
CONST /* Valid account numbers.
  Accounts ={z | 1000<z AND INTEGER(z)};
CONST /* Set of all representable names.
  Names     ={z | CHARSTRING(z)};

INITIALIZATION.
  UNDEFINED.

PROCEDURE file(z). /* Branch query is filed for future processing.
  IN:      z=(id,no,trm,req);
  OUT:     ibuffer'=ibuffer UNION {z};
  EXPT:    NIL;

PROCEDURE transaction. /* A query from the input buffer is processed
                      /* and the answer is placed in the output buffer.
  IN:      z=(id,no,trm,req) AND z MEMBER.OF ibuffer AND
          IF t=(id1,no1,trm1,req1) AND t MEMBER.OF ibuffer
          THEN no<no1+1;

  OUT1:   /* Final buffer states.
  ibuffer'=ibuffer MINUS {z};
  obuffer'=obuffer UNION {(id,no,trm,ans)};

  OUT2:   /* The effect of creating an account for customer c.
  IF req='create',c) THEN
    Records'=Records UNION {(k,c,0)} AND
    ans=(k,c) AND k=NEWACCT;

  OUT3:   /* The result of extracting information about m accounts.
  IF req='read',a[1],...,a[m]) THEN
    IF (a[i]=1,...,m],c[i],b[i]) MEMBER.OF Records
    THEN ans=((a[1],c[1],b[1]),...,a[m],c[m],b[m]));
    ELSE ans='error';

  OUT4:   /* The result of adding signed value v to account a.
  IF req='update',a,v) THEN
    IF (a,c,u) MEMBER.OF Records AND u+v+1>0
    THEN Records'=Records MINUS {(a,c,u)} UNION {(a,c,u+v)}
        AND ans=(a,c,u+v);
    ELSE ans='error';

```

```

OUT5: /* The effect of transferring positive amount v from
/* account a1 to a2.
IF req='transfer',a1,a2,v) THEN
  IF (a1,c1,u1),(a2,c2,u2) MEMBER.OF Records
    AND u1-v+1>0 AND v>0
    THEN Records'=Records MINUS {(a1,c1,u1)} MINUS {(a2,c2,u2)}
      UNION {(a1,c1,u1-v)} UNION {(a2,c2,u2+v)}
      AND ans=((a1,c1,u1-v),(a2,c2,u2+v))
    ELSE ans='error';
EXPT: RESTART;

PROCEDURE z:=retrieve. /* Processed query is removed from the
/* output buffer.
IN: w=(id,no,trm,ans) AND w MEMBER.OF obuffer;
OUT: z'=w AND obuffer'=obuffer MINUS {w};
EXPT: RESTART;

PROCEDURE w:=branchid(z). /* The destination of answer contained in z
/* is determined and returned in w.
IN: z=(id,no,trm,ans);
OUT: w'=id;

BEHAVIOR.
PARBEGIN
  BEGIN /* Database queries are accepted and placed in the
  /* input buffer.
    LOOP {GET(z) FROM ALL ON switch; file(z);}
  END.

  BEGIN /* Database answers are removed from the output buffer
  /* and sent to their sources.
    LOOP {z:=retrieve; w:=branchid(z);
      IF w=1 THEN SEND(z) TO branch1 ON switch;
      IF w=2 THEN SEND(z) TO branch2 ON switch;}
  END.
PAREND.
END-PROCESS database.

LINKS.
/* Definition of logical communication links.
switch: (branch1, branch2, database);
ttylink1: (tty1, branch1);
ttylink2: (tty2, branch2);

```

```
COMMUNICATION.
/* Definition of the communication protocol for each link.
switch: PARBEGIN
    BEGIN LOOP { branch1:SEND[1](z) to database;
                  database:GET[1](z);
                  {branch1:GO[1]; // database:GO[1];} }
    END.

    BEGIN LOOP { branch2:SEND[1](z) to database;
                  database:GET[1](z);
                  {branch2:GO[1]; // database:GO[1];} }
    END.

    BEGIN LOOP { database:SEND[1](z) to branch1;
                  branch1:GET[1](z) from database;
                  {database:GO[1]; // branch1:GO[1];} }
    END.

    BEGIN LOOP { database:SEND[1](z) to branch2;
                  branch2:GET[1](z) from database;
                  {database:GO[1]; // branch2:GO[1];} }
    END.
PARENDS.

ttylink1: UNDEFINED;
ttylink2: UNDEFINED;

END-NET egbank.
```

LANGUAGE DEFINITION

The presentation of the language is organized in a manner similar to that of the DSDL specifications except that the definition of a process is introduced first and is later used in the definition of the net. The discussion of the process consists of a formal statement of the semantics of the process and its component entities (data, procedures, and behavior) and an overview of the language features available for specifying processes. The net and its components (environment, processes, links, and communication) receive a similar treatment.

PROCESS DEFINITION.

The process is the basic functional unit of DSDL, and is similar to the guardian [LISK79] and the monitor [RIDD78] (except that internal parallelism is allowed). It serves to encapsulate data as in the abstract data type [GUTT77, WULF76], and has sole access to its own data. In addition, a process is able to receive and send information via messages as in [HOAR78]. In order to define the functionality of a process, a set of procedures are defined. They perform indivisible operations on data or communicate with the environment. The behavior of a process is then defined as the set of allowable sequences of procedure invocations within the process. The formal definition of the process is shown below.

DEFINITION.

A process p is defined as a five-tuple

$$p = (D_p, T_p, R_p, S_p, B_p)$$

where

$$D_p = (Q_p, H_p, I_p)$$

with Q_p denoting the set of data entities controlled by p , the data invariant H_p being a predicate over Q_p , and the initialization I_p being a predicate defining the initial values for the data in Q_p .

$$T_p = \{z : z = (Ain(D_p, w), Aout(D_p, D'_p, w, w'))\}$$

with T_p representing a set of transformational procedures described by pairs of assertions; the input assertion is a predicate over the data owned by the process p , i.e., D_p , and the input values of the parameters w ; the output assertion is over the old and the new values of the data and of the parameters.

$$R_p = \{z : z = ('true', Aout(D_p, w'))\}$$

with R_p representing a set of message receiving procedures whose (partial) meaning is given by an output assertion which describes the kind of values expected to be received from some other processes.

$$S_p = \{z : z = (Ain(D_p, w), 'true')\}$$

with S_p representing a set of message sending procedures

whose (partial) meaning is given by an input assertion which describes the kind of values expected to be sent to some other processes.

Bp SUBSET.OF (Tp U Rp U Sp)*

with Bp defining the process behavior given in terms of possible sequences of events, where each instance of a procedure invocation is treated as a primitive event.

In the EGBANK example, 'branch1', 'branch2', and 'database' are processes while 'office' is a process type used in defining the first two of the three processes in the net. The basic process definition takes two syntactic forms:

(1) PROCESS process.name;	(2) PROCESS process.name: process.type;
DATA. ... definitions;	PARAMETERS. ... values;
INITIALIZATION. ... initial values;	END-PROCESS process.name.
PROCEDURE ... definition; ...	Note: the process.type has to be declared through the use of the 'DEFINE' statement.
PROCEDURE ... definition;	
BEHAVIOR. ... description;	
END-PROCESS process.name.	

DATA.

The first element in the 5-tuple representing the process p is the data Dp, which belongs to that process. The data is defined as an ordered triple (Qp, Hp, Ip), as shown above. The first element, Qp, is a set of data entities controlled by p. Qp may be of arbitrary complexity and structure and its elements may be accessed only by procedures within the process. The second element, Hp, is the data invariant, which is a predicate describing the properties which must be possessed by the elements of Qp both before and after all data transformations. (See [WULF76] for a discussion of the abstract and concrete invariants used in Alphard). The purpose of the invariant is to provide an aid in checking for preservation of data consistency and to serve as a lemma in proofs concerning the process. The third element, Ip, is a predicate defining the initial values for each data entity in Qp.

The data controlled by some process appears in the "DATA" section of the process definition. Both variables and constants may be declared using statements whose syntax resembles Pascal. The variable declarations are placed side by side predicates that are taken to be parts of the invariant Hp (e.g., VAR n: INTEGER; n>0;). Because of the set theoretical approach to data representation adopted by DSDL, both variables and constants are either sets or elements of sets. Some sets are assumed to

be built-in (e.g., INTEGER) while others are constructed by enumeration (e.g., $S=\{1,2,3\}$), by providing an intensional definition (e.g., $S=\{z \mid 0 < z < 4 \text{ AND } \text{INTEGER}(z)\}$), or by means of standard set operations (e.g., union, intersection, subtraction, cross-product, etc.). In addition to the set notation the standard mathematical notation for functions and relations is also available:

```
CONST f : INTEGER -> INTEGER;
      f(n) <= IF n<1 THEN 1
                  ELSE n X f(n-1);
```

The use of the set theoretical notation is motivated not only by the desire to develop a simple but formal specification language, but also by a deliberate effort to promote a high level of abstraction which, free of unnecessary detail, allows the designer to concentrate on system level issues rather than the design intricacies of its components. Furthermore, most system designers are fairly familiar with set theory, a fact that makes it attractive both from the point of view of ease of use and with regard to the analyzability of the specifications being generated.

PROCEDURES.

The process activities, data transformations and message exchanges, are defined by the procedures it controls. The transformational procedures, given in terms of input and output assertions, describe state changes and return values to be used as input parameters in subsequent procedure invocations. While these procedures are defined by the user of the language, the message exchanges are carried out by two built-in procedures (SEND and GET) whose semantics are stated in the communication section of the net definition. Consequently, their discussion is postponed for now.

The use of nonprocedural specifications in defining the meaning of the transformational procedures enhances the understandability of the process specification. Furthermore, by treating procedure invocations as primitive operations over the data the need for synchronization within a process is avoided in the same way as it is done in the monitor concept employed by concurrent Pascal [HANS77], but without prohibiting concurrency from occurring in the process.

Syntactically, the definition of the transformational procedures is straightforward: pairs of input ("IN:") and output ("OUT:") assertions are used to cover distinct cases; when an input assertion is followed by several output assertions (numbered or not) a conjunction between them is implied; an exception ("EXPT:") assertion may be provided to indicate the action to be taken in case all input assertions fail (e.g., NIL, RESTART, ABORT, etc.); standard predicate calculus is used in constructing the assertions (AND, OR, XOR, NOT, IF-THEN-ELSE, i.e., implication); finally, a name followed by a single quote denotes the value of a data item after the completion of the procedure.

BEHAVIOR.

The behavior of a process is defined as the set of all allowable sequences of events within a process, where an event is defined as an

invocation of a procedure. The following constructs are available for the behavior specification:

```
PARBEGIN concurrent begin-end blocks PARENT  
{ event.sequence1 // ... // event.sequence.n }  
  
BEGIN list of events or event.sequences END  
{ event.sequence1 ... event.sequence.n }  
  
IF condition THEN event.sequence1 ELSE event.sequence2  
  
CASE (condition1) --> event.sequence1  
    ...  
    ()           --> default.event.sequence  
ENDCASE  
  
WHILE (condition) --> event.sequence  
  
LOOP event.sequence  
  
DOUNTIL (condition) --> event.sequence
```

where an event.sequence is

- a procedure invocation followed by a semicolon (e.g., z:=f(a);),
- a list of event.sequences between braces (e.g., {z:=f(a); g(a,z);}),
- a list of concurrent event.sequences (e.g., {{z:=f(a); g(a,z);} // h(a,b);}), or
- any sequence of events described by one of the flow of control constructs listed above.

Because of the nature of distributed processing, in general, and due to the fact that at higher levels of abstraction processes represent entire networks, the availability of both concurrency and nondeterminism in describing the process behavior is essential. The PARBEGIN-PARENT and the concurrent event sequences provide the mechanisms needed to express concurrency, while the CASE construct allows one to support nondeterminism. One last thing to be mentioned here is the scope rules for the variables used in the behavior section of the process definition: their type need not be declared because it is determined by the procedure definitions. The scope rules are the same as in all block structured languages and a mere listing of the variables after the BEGIN or PARBEGIN is required. As an exception, variables that are not declared explicitly are associated with the block in which are first used.

NET DEFINITION.

In order to specify a distributed system, the concept of a net is included in DSDL. A net is defined as a set of independent, concurrent processes which communicate among themselves by means of messages [BELL77, FELD79, RIDD78]. Messages are sent over abstract communications paths

called links. The behavior of these links, along with the behavior of each process, determines the behavior of the net as a whole. The formal definition of the net is given next.

DEFINITION.

A net n is defined as a four-tuple

$$n = (P, P', L, C)$$

where

$$P = \{ p \mid p \text{ is a process} \}$$

$$P' \text{ SUBSET.OF } P$$

with P' representing a set of processes used to model the environment of the system.

$$L = \{ l \mid l \text{ SUBSET.OF } P \}$$

where a link l is defined by the set of processes that may use it.

$$C : L \rightarrow \text{POWERSET.OF} ((\text{UNION OVER ALL } p \text{ OF } (Sp \text{ UNION } Rp))^*)$$

with

$$C(l) \text{ SUBSET.OF } (\text{UNION OVER ALL } p \text{ MEMBER.OF } l \text{ } (Sp \text{ UNION } Rp))^*$$

i.e., $C(l)$ establishes the link behavior which determines the set of allowable sequences of send and receive events over the link.

The first two elements in the 4-tuple describing the net are the set of processes P and a set P' such that $P' \text{ SUBSET.OF } P$. Each of the processes in the set P is an independent unit, and is defined formally in the manner illustrated in the previous subsection. The process is used to model a single processing/storage element in a concurrent and possibly distributed system. These processes represent logical functional units, and not physical processors. Hence, an actual implementation of a process may be split across several real processors or share a single processor with several other processes. As part of the concurrent system, in most cases one or more processes are used to model the environment. These environment processes comprise the set P' .

The last two elements in the net 4-tuple are the set of links L and the communications protocol C . The links connect receiving ports of processes on the link to sending ports of other processes on the link, and represent the available paths of communication within the net. Each link is a logical communications path. Hence, a link may represent a large number of physical connections (such as paths through a packet switching network) or simply a message buffer in shared memory. For each link l a set $C(l)$ of allowable sequences of send and receive type events in the processes that it connects is defined. This set essentially describes the communications protocol on the link, and will be referred to as the link behavior. The events used to define each link behavior are all of the send and receive type events in the processes which it connects, and the link behavior itself is specified using constructs introduced earlier for

defining a process behavior.

ENVIRONMENT.

In general, the nature of the environment with which a system is intended to interact affects the design not only with respect to the functionality that needs to be supported but also in terms of the workload characteristics. Systems having identical functionality may exhibit significant differences in design complexity due to the distinct assumptions made about their environment. Consequently, a system specification can hardly be considered complete unless these, often hidden, assumptions are made explicit. In DSDL, processes declared to be "EXTERNAL" encapsulate the nature of the environment. However, whenever the environment plays only a marginal role in the specification of the system, the external processes and the links that connect them to the system may be declared to be undefined. Under such circumstances, the environment is presumed to provide the system with "appropriate" messages on demand and to immediately accept all messages generated by the system.

PROCESSES.

The processes that form a net are defined in the manner already discussed above. It must be added, however, that processes at one level of the specification may represent abstractions of entire nets to be identified later. DSDL allows one to state this fact through the use of the attribute "REFINEMENT.OF" as in the example below:

```
NET netname; REFINEMENT.OF pname.  
...  
END-NET netname.
```

where the net "netname" is identified to be a refinement of the process "pname". Furthermore, any entity in the net may be declared to be a refinement of some entity in the process as long as consistency is preserved. Unfortunately, general consistency proof techniques are still under investigation and ad-hoc methods are used instead.

LINKS.

Each link is defined by its unique name, the processes that may use it, and a description of its behavior given in the section on communication. More than two processes may have access to the same link and the same two processes may have more than one link in common. The motivation for this approach is to be found in the desire to enable the description of arbitrary interconnection structures. Furthermore, because links are logical and not physical in nature, a link may be later refined as a net that implements the behavior of the respective link. A packet switching net, for instance, may be described first as a link between all the nodes it services and may be subsequently refined to include the switching nodes and their protocols.

COMMUNICATION.

For every link in the net, a behavior description has to be included in the communication definition section. The link behavior defines the communication protocol associated with the respective link, i.e., the semantics of the GET and SEND commands. In defining the link behavior, the designer may use the same the same means of specification as in the

description of a process behavior except that the set of events that may be involved is restricted to

- invocation of a receiving procedure
(e.g., branch1:GET[1](z) from database;)
- invocation of a sending procedure
(e.g., branch1:SEND[1](z) to database;)
- resumption of processing after the invocation of a sending or receiving procedure (e.g., branch1:GO[1];)

in processes associated with the respective link.

(Note: the number between the square brackets serves the purpose of matching resumption of processing events with corresponding invocations of sending and receiving procedures.)

In the definition of the behavior of the link called "switch", for instance, the following BEGIN-END block appears:

```
BEGIN LOOP { branch1:SEND[1](z) to database;
              database:GET[1](z);
              {branch1:GO[1]; // database:GO[1];} }
END.
```

It establishes the fact that once a SEND is invoked, the issuing process waits for completion of the corresponding GET and that no GET is invoked by the database unless the corresponding SEND has been issued first. Moreover, after exchanging the message "z", both processes may resume processing in no particular order. Similar protocols are described for the other message exchanges occurring over the link "switch".

The behavior of the net as a whole is determined by the behaviors of its processes and links. The net behavior is formally defined as the set of all sequences of events that have the property that are consistent with the local behavior of each of the processes and links:

$$B = (Bp_1 \% \dots \% Bp_n) \% (C(11) \% \dots \% C(1m))$$

where

B is the net behavior
B_{pi} is the behavior of process i (for i=1,...,n)
C(1j) is the protocol for link j (for j=1,...,m)

and

% is the synchronized shuffle operator defined as follows

$$\begin{aligned} V \% W &= \{ v \% w \mid (v \text{ MEMBER.OF } V) \text{ AND } (w \text{ MEMBER.OF } W) \} \\ a \% b &= \{ab, ba\} \\ a \% a &= \{a\} \end{aligned}$$

CONCLUSIONS

There are four key features that make DSDL an attractive candidate for a distributed system design language: high degree of formality, designer specified communication protocols, non-procedural character, and strong emphasis on separation of concerns.

The high degree of formality is achieved through the use of set theory and predicate calculus. Both are a common place background for recent computer science graduates and for most experienced system designers. Furthermore, sets and set operators are already present in languages such as Pascal while predicate calculus is central to current work in artificial intelligence. Thus, it appears that future attempts to support and analyze DSDL may require initially no development of new technology but use of already available expertise in the computer science field.

The freedom of specifying arbitrary communication protocols is an essential feature for any distributed system design language. One can hardly expect a designer to have to have to limit the design space due to specification language limitations. Moreover, evaluation of alternate communication protocols is an important design activity and ought to be supported in a straightforward manner, i.e., one should be able to alter the communication protocols without having to consider changes in the other aspects of the system specifications.

Advances in proofs of the correctness of sequential programs have been based on the non-procedural nature of I/O assertions. It is our conjecture that the specification of distributed systems could benefit from the use of I/O assertions for the description of presumably sequential activities within a process, and from the extension of the non-procedural type of specifications to behavior specification. (Efforts to accomplish the latter goal have not been completely successful and, consequently, the current definition of DSDL describes both behavior and communication in a procedural manner.) Furthermore, non-procedural specifications avoid the addition of extraneous detail during the different stages of the design and specification.

It is a generally accepted fact that hierarchical design is useful in reducing the overall complexity of large systems. Further reductions in the complexity of the specification are achievable by imposing some appropriate structure over each level of the hierarchical specification. In DSDL, this is achieved by applying the principle of separation of concerns in such a way as to assist in the logical verification of the specification at that level. The solution is to structure the design specifications based on correctness and self-consistency proof dependencies. In DSDL, for instance, one would first prove the preservation of specified invariants over the data local to a procedure. Proofs about data may then be employed as lemmas in proofs about the procedures; proofs about procedures may be used as lemmas in proofs about the processes, etc. The dependence of the higher-level entities on the lower-level entities creates the opportunity for simplification of proofs about the system as a whole through the use of hierarchically structured proofs.

DSDL has been exercised on several small problems. These exercises were useful in demonstrating the language's power of expression. Nevertheless, there are many unresolved issues. First of all, a meaningful evaluation of the language demands its use on a real-life project of adequate complexity. Second, future advances in the study of the formal aspect of the language must be carried out in preparation for potential incorporation of DSDL in a computer-aided design system. As of now, a definition for consistency between levels has been proposed but proof strategies need to be developed. Finally, a complete system specification language ought to include also the capability to define processors and their characteristics, the rules for allocating processes among processors, and performance specifications. Research in these areas is currently under way and its results will be reported elsewhere.

REFERENCES

- [BELL77] Bell, T. E., Bixler, D. C. and Dyer, M. E., "An Extendable Approach to Computer-Aided Software Requirements Engineering," IEEE Trans. on Soft. Eng. SE-3, No. 1, pp. 49-60, January 1977.
- [BOOT80] Booth, T. L. and Wiecek, C. A., "Performance Abstract Data Types as a Tool in Software Performance Analysis and Design," IEEE Trans. on Soft. Eng. SE-6, No. 2, pp. 138-151, March 1980.
- [CAMP79] Campbell, R. H. and Kolstad, R. B., "Path Expressions in Pascal," Proc. 4th Int. Conf. on Soft. Eng., pp. 212-219, 1979.
- [FELD79] Feldman, J. A., "High Level Programming for Distributed Computing," CACM 22, No. 6, pp. 353-368, June 1979.
- [GUTT77] Guttag, J., "Abstract Data Types and the Development of Data Structures," CACM 20, No. 6, pp. 396-404, June 1977.
- [HANS77] Hansen, B., The Architecture of Concurrent Programs, Prentice-Hall, 1977.
- [HOAR78] Hoare, C. A. R., "Communicating Sequential Processes," CACM 21, No. 8, pp. 666-677, August 1978.
- [LISK77] Liskov, B., et al, "Abstraction Mechanisms in CLU," CACM 20, No. 8, pp. 564-576, August 1977.
- [LISK79] Liskov, B., and Berzins, V., "An Appraisal of Program Specifications," Research Directions in Software Technology, P. Wegner, editor, MIT Press, pp. 276-301, 1979.
- [RIDD78] Riddle, W. E., et al, "Behavior Modeling During Software Design," IEEE Trans. on Soft. Eng. SE-4, No. 4, pp. 671-678, July 1978.
- [ROBI77] Robinson, L. and Levitt, K. N., "Proof Techniques for Hierarchically Structured Programs," CACM 20, No. 4, pp. 271-404, April 1977.
- [ROSS77] Ross, D. T., "Structured Analysis (SA): A Language for Communicating Ideas," IEEE Trans. on Soft. Eng. SE-3, No. 1, pp. 16-34, January 1977.
- [SANG79] Sanguinetti, J., "A Technique for Integrating Simulation and System Design," Proc. Conf. on Simulation, Measurement and Modeling of Computer Systems, pp. 163-172, August 1979.
- [SMIT79] Smith, C. and Browne, J. C., "Modeling Software Systems for Performance Predictions," Proc. Computer Measurement Group X, pp. 321-341, December 1979.

[WULF76] Wulf, W. A., London, R. I., and Shaw, M., "An Introduction to the Construction and Verification of Alphard Programs," IEEE Trans. on Soft. Eng. SE-2, No. 4, pp. 243-265, December 1976.

APPENDIX H
MODERN PROGRAMMING ENVIRONMENT ASSESSMENT

H.1. INTRODUCTION

H.1.1 OBJECTIVES

This document provides an assessment of the Modern Programming Environment (MPE) being developed for the Defense Mapping Agency (DMA). The MPE effort under review is a design study performed under contract F30602-81-C-0039 to Rome Air Development Center (RADC) for DMA. The assessment of that study, as documented in this report, is based solely on government supplied documents and information gathered during meetings with persons involved in the MPE program. This information is evaluated from the perspective of the Total System Design (TSD) Facility.

The rationale for this assessment lies in the fact that the MPE can be viewed as a TSD facility specialized to the production of software at DMA. As a result, many of the issues considered in defining the TSD facility are relevant to the MPE. The objectives of the assessment are:

- (1) To evaluate the current MPF plans and, if needed, prepare alternatives.
- (2) To identify issues which should be considered in future MPE efforts.

The assessment results pertinent to objective (1) are presented in section H.1.2, and are summarized in Figure H-1. The results pertinent to objective (2) are presented in Section H.5. As the MPF study was not complete at the time of this assessment, the documents and personal communications which served as the basis of this assessment must be considered preliminary in nature. As a consequence, it is possible that some of the issues raised in this document may be resolved prior to completion of the MPE study.

H.1.2 RECOMMENDATIONS

H.1.2.1 OVERVIEW

The recommendations presented in this section are concerned mainly with the near term MPE development effort. We believe that the tasks associated with Phase II of the development will be greatly affected by the results of the near term (Phase I) development, and so specific recommendations concerning Phase II are not presented. In addition, as the near term effort will provide the basis for all long term facility development, the overall success of the MPE depends largely on the successful implementation and phase in of the near term MPE facility and its associated software development methodology. The recommendations presented are intended to reduce the risk of this near term development.

TASK	PHASE I	PHASE IA
FACILITY DEVELOPMENT	<ul style="list-style-type: none"> *Design and implementation for the near term experimental system. *Training for the near term experimental system. *Design of the near term full scale system. *Development of phase in plans for the near term full scale system. 	<ul style="list-style-type: none"> *Implementation of the near term full scale system. *Performance of the phase in plans for the near term full scale system.
METHODOLOGY DEVELOPMENT	<ul style="list-style-type: none"> *Definition of the MPE software development methodology. *Development of requirements for enhancements to the phase II MPE. *Development of phase in plans for the methodology. 	<ul style="list-style-type: none"> *Performance of the MPE software development methodology phase in.
R&D PREPARATION FOR PHASE II START UP	<ul style="list-style-type: none"> *Assessment of the potential use of the project database concept in the far term MPE. *Evaluation of techniques for achieving high levels of integration in the MPE tool set. *Evaluation of facility structures which enable smooth facility evolution and responsiveness to technological changes. *Development of technical and managerial procedures for assuring smooth evolution of the MPE. 	

Figure H-1. ALTERNATE PROPOSAL FOR PHASE I/IA MPE DEVELOPMENT

Based on our evaluation of the current MPE plans and on the issues presented in Section H.5, the following tasks are proposed as a framework for the MPE development effort (see Figure H-1).

- Facility Development
- Methodology Development
- R&D Preparation for Phase II Startup

Each of these tasks is independent of the others and involves a different area of expertise, allowing them to be carried out separately. The separation of the Facility Development and the Methodology Development tasks is highly recommended as both are essential to the initial success of the MPE and independent scheduling would preclude the compromising of one task due to time or manpower shortages in the other. A more detailed treatment of each of these tasks is presented below.

H.1.2.2 FACILITY DEVELOPMENT

The facility development should proceed largely as planned in the MPE study, except that no methodology related development should be included. In particular, we agree with the selection of the VAX as the basis of the facility development, as the future availability of new software tools and specialized hardware support for this system appears to be excellent. The tool set also appears to be satisfactory for initial development. Special emphasis should be placed on the plans for a phased introduction of the MPE into the DMA working environment, as this process will have a strong bearing on the initial success of the MPE. The basic tasks to be carried out are as follows.

Phase I

- (1) Design and implementation for the near term experimental system.
- (2) Training for the near term experimental system.
- (3) Design of the near term full scale system.
- (4) Development of phase in plans for the near term full scale system.

Phase IA

- (1) Implementation of the near term full scale system.
- (2) Performance of the phase in plans for the near term full scale system.

H.1.2.3 METHODOLOGY DEVELOPMENT

The methodology development should be carried out based on a review of current DMA standards, the DMA environment, and the MPE tool set. The methodology established should completely define the phases of the life cycle (in a manner similar to that in the current DMAAC standards), the documents produced by each phase, the tools used to develop the documents, document organization standards, and tool usage techniques which support these standards. In addition, the methodology should establish review points in the development process, the scope of the reviews, and the tools which will be used to support the review process at each point. The basic tasks to be carried out are as follows.

Phase I

- (1) Definition of the MPE software development methodology.
- (2) Development of requirements for enhancements to the Phase II MPE which are needed to support the methodology. These requirements should identify improvements to existing tools, new tools which will provide more complete coverage of the activities identified by the methodology, and requirements for the integration of the tools based on the usage patterns established in the methodology.
- (3) Development of phase in plans for the methodology.

Phase IA

- (1) Performance of the MPE software development methodology phase in.

H.1.2.4 R&D PREPARATION FOR PHASE II STARTUP

This task is concerned with carrying out research and development activities which are beyond the scope of the facility and methodology development tasks, but which are required for the startup of the Phase II development effort. Many of the concerns to be addressed by this task have been identified in Section 5 of this assessment. A sampling of these R&D tasks includes:

- (1) Assessment of the potential use of the project database concept in the far term MPE.
- (2) Evaluation of techniques for achieving high levels of integration in the MPE tool set.
- (3) Evaluation of facility structures which enable smooth facility evolution and responsiveness to technological changes.
- (4) Development of technical and managerial procedures for assuring smooth evolution of the MPE.

These R&D tasks will lead to the development of a preliminary design for the far term MPE system.

H.1.3 TECHNICAL SUMMARY

H.1.3.1 CURRENT DMA SOFTWARE DEVELOPMENT NEEDS

The software development effort at DMA can be broken down into two classes. The first class consists of software developed and maintained within DMA. This software can be further categorized as closed shop products (for a controlled environment on the production mainframes) and open shop products (not for the controlled mainframes). The second class consists of software contracted to outside organizations but maintained within DMA.

In order to keep pace with the ever increasing demand for software, DMA must improve the productivity of the software development and maintenance process, and improve the quality of the software produced. In order to make these improvements, several needs must be met. First, automated support for software development and maintenance activities is needed to make efficient use of personnel and aid in the detection of errors and inconsistencies. Second, project management aids are needed to improve the visibility of development and maintenance activities and to aid in project control, scheduling, and resource allocation. Finally, an appropriate methodology for software development and maintenance is needed, along with verification and enforcement aids for the methodology.

H.1.3.2 MODERN PROGRAMMING ENVIRONMENT OVERVIEW

The MPE is concerned solely with the needs of the product oriented software developed or maintained within DMA. It does not specifically aid in the development or maintenance of the large GIP systems mentioned above, as these systems are not as yet part of the main software production environment at DMA. In addition, the MPE is oriented largely to the needs of the closed shop software development. The two main functions to be served by the MPE are described below.

The first function of the near-term MPE (referred to simply as the MPE) is to improve the productivity of development and maintenance, improve the quality of the resulting software and documentation, and improve the visibility and control for management. This is accomplished by providing:

- a set of automated tools to support development activities and project management over the entire software life cycle;
- a methodology for software development based on the tools provided and the DMA environment; and
- support for the training of DMA personnel.

The second function of the MPE is to provide a basis for long term facility development. The MPE is designed to provide an elementary facility in which tools, usage techniques, and methodologies can be tried and analyzed. The experience gained through use of the MPE will then serve as a guide for future facility development. In addition, the gradual development and enhancement of the initial MPE facility will allow

for the incremental training of DMA personnel.

H.1.3.3 TOTAL SYSTEM DESIGN FACILITY OVERVIEW

As part of the TSD study, a high level specification for a system development facility called the TSD Facility was developed, consisting of an automated system to provide a software development environment (SDE) and a set of resources which are required to support its operation and use. What makes the TSD Facility concept unique is the fact that, rather than revolving around a particular set of tools or specification languages, it assumes a functionalist point of view. Factors considered in the TSD Facility concept include the dynamics of tool development, the relationship between organizations and facilities, the impact of project management objectives, geographic distribution, multi-organization project management, portability, technology transfer mechanisms, specialization, etc. The emphasis is placed on the evolutionary process to which the facility will be subjected and on facility structures that will be responsive to evolving needs.

H.1.3.4 MODERN PROGRAMMING ENVIRONMENT ASSESSMENT

The MPE design, as proposed in MPE study, was assessed on the basis of the technical and managerial environment at DMA and the concepts developed in the TSD study. Since the TSD study was directed at the general problem of designing computer based systems, the TSD view of a support facility is of a wider scope than that taken in the MPE study. Each aspect of the MPE design was evaluated from this broader perspective and with respect to the specific needs of DMA. Based on this evaluation, the proposed MPE design appears to be satisfactory for the near term MPF development. Many issues were identified, however, which should be addressed during the MPF development process. These issues are discussed fully in Section H.5.

H.1.4 INFORMATION SOURCES

Interactive Computer Program Development System Study Final Report,
Contract Number F30602-81-C-0039

Interactive Computer Program Development System Study
Functional Description, Contract Number F30602-81-C-0039

Interactive Computer Program Development System Study
System/Subsystem Specification, Contract Number F30602-81-C-0030

Software Life Cycle Standards, Defense Mapping Agency Aerospace Center

Tutorial: Software Development Environments, IIEEE Computer Society

H.2. CURRENT DMA SOFTWARE DEVELOPMENT NEEDS

H.2.1 CURRENT PROCEDURES

The software development work at DMA can be divided into two categories, each with different support needs. The first category is comprised of new software which is developed from scratch. Programs in this category are usually developed for a single Mapping, Charting and Geodesy (MC&G) product application, and their development may take place in either an open shop or closed shop environment. The second category is comprised of the maintenance and enhancement of existing software developed at DMA, and the maintenance of software developed by outside contractors. This second category represents the major portion of the software effort at DMA.

The method used by DMA to develop software can be characterized as follows. In the requirements definition portion of the life cycle, requirements specification generally is informal and has no automated support. Some efforts are under way at DMAAC, however, which use formal requirements specifications that are generated by hand. In the area of program design, there is no formal specification technique and no automated support. Some organizations, however, do make use of some form of program specifications. Programming itself is performed mostly in dialects of FORTRAN and COBOL, although some assembly language programming is also done. Automated support for this process is provided by the language compilers. In the area of program testing, some automated tools are in existence within DMA to support this process. However, these tools are not in general use. Finally, the major effort over the system life cycle is spent in the maintenance and enhancement of programs. Within DMA, this process is ad hoc with no automated support to control and document modifications.

H.2.2 DMA NEEDS

In order to guide the MPE design, the perceived DMA needs for software development and maintenance support were studied by General Dynamics. An informal life cycle model consisting of phases for requirements definition, program design, coding, testing, production, and maintenance was used, and needs within each category were investigated. In addition, the needs for project management and training of personnel were also investigated. The specific needs which have been documented by General Dynamics in each area are presented below.

Development and Maintenance Needs

- Reevaluation of current methodologies based on the availability of automated support.
- A computer supported requirements specification language.
- A computer supported program design language.
- A program prototyping capability.
- Support for production code optimization.
- Configuration control and requirements tracking.

Management Needs

- Project scheduling aids, such as schedule impact analysis and resource allocation (especially manpower allocation).
- Project review aids, including improved milestone identification and the establishment of quality assurance procedures and guidelines.
- Project history statistics to provide information for the management of future projects.

Personnel Needs

- Interactive access for all personnel.
- Graphic display capabilities.
- Natural language interface.
- Modern data entry techniques.
- Rapid turn-around time.
- A decrease in the volume of paperwork.
- A training/orientation program.
- A user assistance service to aid personnel in solving system usage problems.

H.3. MPE OVERVIEW

H.3.1 MPE OBJECTIVES

The main objective of the MPE is to meet, as much as possible, the current software development and maintenance needs of DMA. These needs revolve around three main concerns. First, DMA must improve the productivity of its software development and maintenance in order to meet current and future demand for its MC&G products. Second, DMA must also improve to quality of its software and documentation in order to help control the ever rising cost of program maintenance. Lastly, the visibility of the entire development and maintenance process must be improved so that management will have better control over the entire process.

The MPE facility was designed in order to meet these needs at DMA. To accomplish this, the facility will provide the following: (1) a set of automated tools to support software development, maintenance, and project management activities; (2) a methodology for the use of these tools and the facility as a whole; and (3) training for DMA personnel in the proper use of the facility.

H.3.2 MPE DESCRIPTION

As part of the MPE facility design, the following equipment configuration has been proposed. Central to the MPE equipment configuration is the Tool Bearing Host (TBH), which will support the automated tools relating to software development, maintenance, and project management. The TBH function is to be provided by a VAX 11/780 minicomputer, with the requisite disk and tape mass storage equipment and a number of interactive terminals to serve as workstations. In addition to the TBHs, the existing production mainframes also have a role in the MPE. The mainframes are to be connected to the TBHs through a communications link to allow for the transportation of production programs from the development to the production environment.

The set of development and maintenance tools selected for the MPE consists of the following.

- USE.IT is a tool which supports the high level design and documentation of software. With the aid of a library of defined functions, USE.IT is also capable of producing high level language code directly.
- SDDL (Software Design and Documentation Language) is a language and associated analysis tools for detailed program design specification and documentation.
- FORTRAN 77 and COBOL 74 compilers are the major high level language processors to be supported.
- FAVS and CAVS (FORTRAN Automated Verification System, COBOL Automated Verification System) are to provide support for program testing. Static and dynamic analysis of program usage and path coverage is performed.
- IS/1 (Interactive Systems/One) is a compatibility package which

makes many of the UNIX Programmer's Work Bench (PWB) tools available on the VAX 11/780. These tools include text processing and configuration control.

The management support functions are to be provided by a single tool called VUE. The focus of VUE is on providing support for project planning and control activities. Its analysis is based in established networking techniques, and provides support for the following: (1) time analysis, (2) cost analysis, (3) resource analysis, (4) resource allocation, and (5) report generation.

Lastly, the training function is to be supported by a tool called HYPERGRAPHICS, which is to be hosted on a microcomputer system for portability. This tool will enable low cost training of DMA personnel outside of the production environment. The basic function of the tool is to support the preparation and presentation of lecture material, allowing for structured movement through the lecture material and the execution of example programs within the framework of the training tool.

H.3.3 MPE USAGE

Although a formal methodology has not yet been developed for the MPE, the manner in which the MPE might be used to develop programs can be characterized as follows. USE.IT will be used to develop the high level design of a program (which is referred to as the "requirements definition" in the MPE study). Reports describing the design and presenting some analysis are automatically produced for management review. If the program is simple or is composed of predefined library functions, then the high level language code for the program can be generated automatically by USE.IT. (The classes of programs which can be developed automatically by USE.IT should be determined during the Phase I development effort.) For the detailed design of the more complex programs or for simple modifications to existing programs, SDDL will be used. Design reports (Software Design Documents) are produced automatically. Once the design has been completed, the programs are then written and compiled using the FORTRAN 77 or COBOL 74 compilers. Programs (either produced manually or generated automatically by USE.IT) are tested with the aid of FAVS or CAVS, which produce a number of reports on the testing status. These reports can then be reviewed by project management before authorizing a program for production status. Once authorized for production, a program is then transported to the production mainframes, where it is compiled and run. In addition, all documentation and source code for the program are brought under configuration control using the Source Code Control System (SCCS) component of the IS/1 package. All subsequent maintenance and enhancements performed on the program are then done with the help and under the control of SCCS.

H.4. TSD FACILITY OVERVIEW

H.4.1 THE TSD PERSPECTIVE ON SOFTWARE DEVELOPMENT

The TSD facility concept has been developed to encompass all of the current trends in software development environments, and more. In its simplest terms, a TSD facility consists of both an automated system to provide a software development environment, and a set of physical resources which support its operation and use. What makes the TSD facility concept unique is that fact that, rather than revolving around a particular set of tools or specification languages, it assumes a functionalist point of view. In contrast, current SDMs tend to be organized around a particular language (Ada, LISP), operating system (UNIX), or methodology (DREAM, HDM). Factors considered in the TSD facility concept include the dynamics of tool development, the relationship between organizations and facilities, the impact of project management objectives, geographic distribution, multi-organization project management, portability, technology transfer mechanisms, specialization, etc. The emphasis is placed on the evolutionary process to which the facility will be subjected and on facility structures that will be responsive to evolving needs.

The discussion in the remainder of this section revolves around three main concepts, namely those of Environment, System, and Facility. An Environment is a user view of the services and data provided by a system. A System is the entity which provides an environment or set of environments for its users. Lastly, a Facility is everything which is needed to support a System and its usage at a particular location.

H.4.2 THE TSD ENVIRONMENT CONCEPT

The TSD Environment, which represents the user's view of the system, consists of the following items. The first item is the set of tools which are provided to the user. In the TSD view, the tool set should be well integrated and cohesive, which can be achieved best through their organization around a central project database. The next item is the set of project data available to the user, which determines his view of the project he is working on. The last item is the user interface, which determines the method of access to the tools and data, as well as the method of interaction. This interface should be as friendly and supportive as possible, in order to make efficient use of the users' time.

The tools provided within an environment must be integrated and support those activities for which the environment is specialized. One category of tools, which is present in all environments, is that of the core tools. Core tools provide general services to the user, such as general text processing, database manipulation, or configuration control. In order to be general, these tools must be language and application independent. While there may be some dependency of the tools on the form (schema) of the data which they are accessing, they must be independent of the semantic content of the data (for example, the IS/1 core tools all assume a fixed schema, that of a continuous stream of characters). The

second category of tools identified within TSD is that of the specialized tools, which are tailored to specific languages, applications, or techniques. Some examples of specialized tools are management tools such as cost estimation (special application), requirements specification analysis tools (special language), design specification analysis tools, and test coverage analysis tools. Those tools which have been selected for an environment must be integrated, i.e. must be able to work with each other within a common user interface structure. Two particular techniques for integrating a tool set are the organization of the tools around a central project database, and the support of automatic transition from one tool to another.

The user interface presented in the environment should be designed in order to make efficient use of the users' time. A basic assumption of TSD is that this interface will be provided through interactive workstations, in order to allow immediate feedback to the user. All tools which the user must use should be accessible through a common command language, which should have the flavor of natural language, without strict syntactic constraints. The interface should also assist the user as much as possible in learning and using the interface, by providing such services as online documentation. Lastly, it should be possible for individual users to tailor the interface to their own particular needs.

The set of project data available to the user within an environment is assumed under TSD to be maintained and controlled by means of a central project database. This database serves as a central repository for all information relating to an individual project. Aside from maintaining each individual environment's view of the project data, the database serves as a focal point around which the environment tool set is organized. A particular advantage of this organization is that it makes it easier to develop tools to analyze the consistency between different portions of the project data, such as the consistency between the requirements specification and the design.

A final aspect to be considered in the discussion of the TSD environment concept is that of the control of multiple environments within a single project. Each environment has its own set of tools and capabilities with respect to tool usage and project data which is available. Through control of the tools and capabilities allocated to an environment, the environment can be specialized (and restricted) to a particular application. Particular criteria for specialization are project type, phase of a project (requirements definition, design, programming, maintenance), and role of the user in the project (management, development, technical review). An environment manager for a project can create sub-environments whose tools and capabilities are a subset of his own (hence environments are hierarchically structured). Several environments may be created around the same set (or common subsets) of project information, allowing for many views of the project to exist simultaneously. Within each environment, the user may manipulate his portion of the project data without affecting the views of the other users. Once he has completed his work (such as a programmer completing the coding of a module), any new project data generated can be reviewed by management before incorporating it into the global project view. Finally, through control of the tools and data available to individual users, the

entire project database can be protected from unauthorized access and manipulation.

H.4.3 THE TSD SYSTEM CONCEPT

A TSD System is any system which implements a TSD Environment structure as characterized above. As part of the TSD study, the high level design of a TSD System was carried out in order to illustrate a method of implementing the various environment concepts. An overview of that design is presented below.

Figure H-2 is a top-level diagram of the proposed TSD System design which incorporates all of the aspects of the TSD Environment described above. The main control module of the system is the Command Language Interpreter (CLI). All user communication must pass through this module for interpretation, control, and possible execution. Hence the CLI presents the main user interface for the system. The CLI keeps track of all system resources and user authorizations, allowing it to automatically provide users with their proper environment when they log onto the system.

User communication with the TSD System is through the interactive work stations (WS₁,...,WS_n). These stations may be as simple as dumb CRT terminals, or as complex as sophisticated graphics work stations, depending on the user/application requirements.

Each TSD System may also be connected to selected specialized peripherals (SP₁,...,SP_m). Although every installation will have standard peripherals, such as line printers and tape drives, more specialized devices such as those needed for graphics display or sophisticated data gathering may not be available so universally. In order to share the use of such devices, one of the specialized peripherals is assumed to be a network connection. This allows a user from one TSD System to access and use the specialized services that may be available at another installation. As indicated in Figure H-2, all such accessing goes through the appropriate interface routines (T₁,...,T_m), but the central control still resides in the respective CLI modules.

All communications with the project database must pass through the Database Access Control (DAC) module. The DAC uses the appropriate project database tables to either allow or disallow the requested user/tool access. This is the mechanism through which the control of user access to information inherent in the environment concept is performed. In order to provide for portability these access requests assume a standardized TSD System database structure. However, this is essentially a "pseudo" or "logical" database in that it probably will be more economical to use an existing commercial database system for the actual physical data storage and retrieval operations. Thus the TSD database management system may be viewed as an interface between the assumed TSD logical database and the actual physical database. Note that this means that different commercial database management systems could be used at different TSD System installations, with the only added expense that of redefining the TSD DBMS interface.

(Work Stations Offering Specialized
Design and Management Environments)

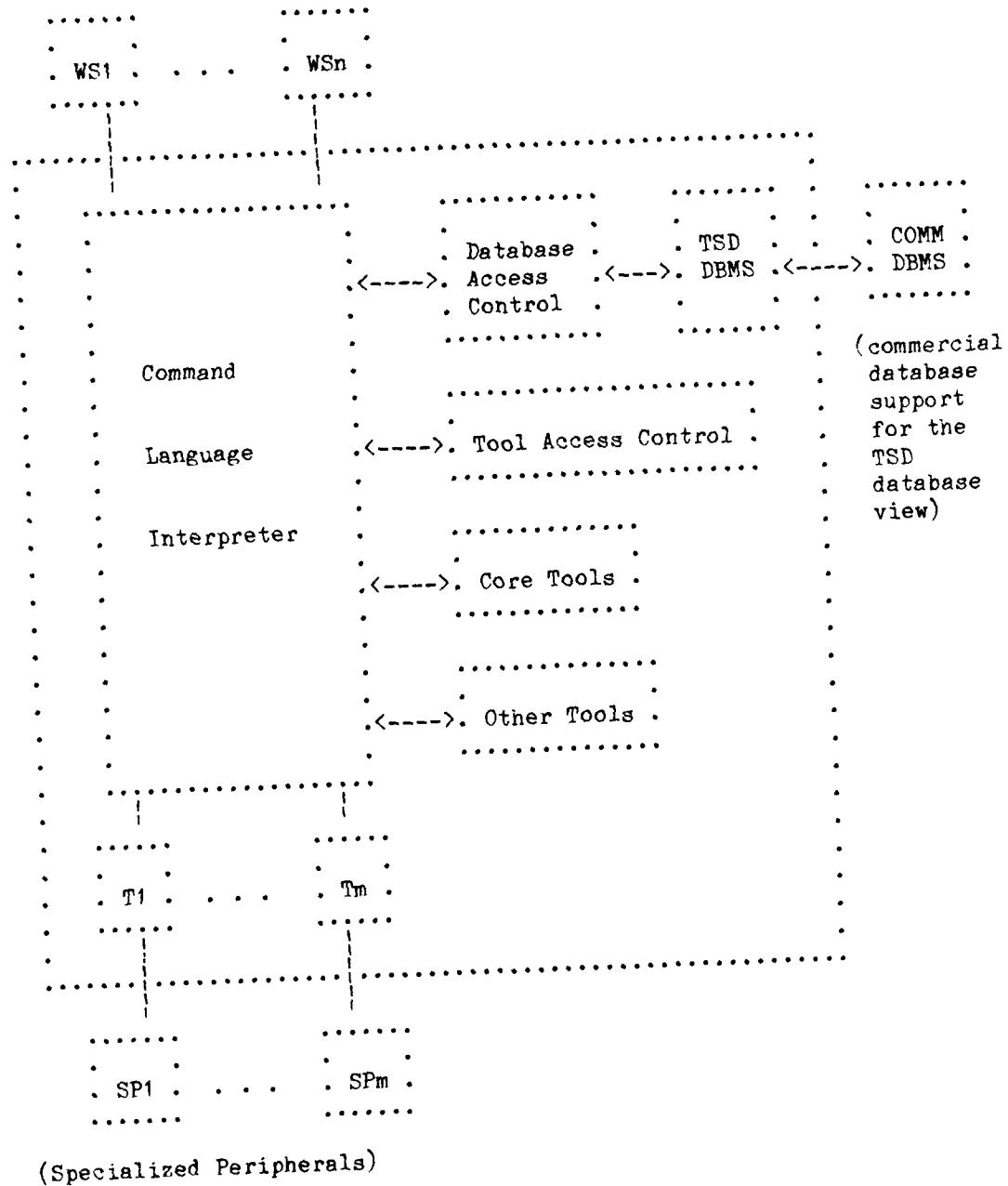


Figure H-2. TSD SYSTEM HIGH LEVEL DESIGN

The CLI uses the Tool Access Control (TAC) module to maintain control over the use of the tool set. User requests for tools, tool requests for tools, tool compatibility and interface requirements, and all other information about tool usage will be maintained through this module. Hence, it provides the mechanism through which tool usage is restricted for individual environments. In addition, the TAC maintains the integrity of the system integration through the enforcement of tool compatibility and interface requirements. New tools to be added to the system are integrated into the system by supplying the appropriate information to the TAC module.

The Core Tools module represents the collection of all standard TSD System tools available at any given time. As mentioned previously, particular applications will need specialized tools or utilize specialized peripherals that require unique tools. In general, these tools are contained in the Other Tools module, which will vary in content from system to system.

H.4.4 THE TSD FACILITY

A TSD Facility is a TSD System implementation along with everything needed to support its operation and use. In particular, the support for the System consists of a set of physical resources, a methodology for using the facility, and facility personnel. A more detailed description of each of these aspects of the Facility is presented below.

The center is the TSD Facility is, of course, the TSD System which it provides for its users. Typically, a TSD System can be implemented in software on top of a standard commercial operating system along with a standard commercial database. In addition, some specialized hardware (such as graphics displays) may be needed to implement some functions of a particular system. Any tools provided in excess of the core tool set may be specialized according to the overall purpose of the facility.

The physical resources provided for the Facility must be sufficient to meet the needs of the TSD System implementation, the Facility staff, and the Facility users. Some examples of resources required by the facility are computer equipment (processors, memory, mass storage, terminals, specialized hardware, etc.), building space, and office equipment.

Another basic component of a TSD Facility is the methodology which specifies how the facility and its tools are to be used to solve problems. This methodology must present a well defined life cycle for the products being developed by the Facility, including a description of the documents and tools involved at each point in the life cycle. Also important are the procedures for verifying and enforcing compliance with the methodology. To aid in the usage of the methodology, the TSD System should be specialized to support it directly. In particular, specialized tools should be selected which aid in and enforce the use of the methodology. Also as an aid in usage, a set of guidelines for tailoring the methodology to specific applications should be provided.

The last basic component of the TSD Facility is the Facility staff, which can be divided into technical staff and management staff. The technical staff provide all of the technical support needed by the TSD System and its users. Their functions include operation of the TSD System implementation, assisting users in performing System related activities, training of new Facility users, and performing development and maintenance functions for the facility. The management staff ensure that all aspects of the Facility operation proceed correctly and efficiently. The functions of this staff include the coordination, supervision, planning, and monitoring of the Facility and its usage.

H.5. MPE ASSESSMENT

H.5.1 OVERVIEW

The MPF design presented in the MPF study was assessed based on the environment at DMA and the concepts developed in the TSD study. Since the TSD study was directed at the general problem of designing complete hardware/software systems, the TSD view of a development facility is of a wider scope than that taken in the MPF study. Each aspect of the MPE was evaluated from this broader perspective and with respect to the specific needs of DMA.

Based on this evaluation, the current MPF design was determined to be a satisfactory basis for the MPF development. Many issues were identified, however, which did not fall within the scope of the MPE study but which should be taken into account prior to the phase II development effort. The purpose of this section is to present a discussion of these issues within the context of the TSD overview of the previous section, and to discuss the rationale for our alternative proposal for the phase I/IA MPF development.

H.5.2 FACILITY DEVELOPMENT CONCERNS (PHASE I/IA)

Maturing of the tool set. In order to minimize cost and development time, it was specified that the tool set for the MPF consist mostly of mature tools, i.e. tools which already exist and have been successfully used on a non-trivial basis. The tool set selected for the MPE has largely met this requirement, although some tools are relatively new and may require some development to become fully mature. One such tool is USE.IT, which has had limited usage in a practical software development environment and which requires a rich library of program modules tailored to the DMA application environment in order to be effective. Another major area requiring development is the interfacing of the non-UNIX tools with the IS/1 system, which will be a major task of the MPF development.

User interface. The major user interface issues are centered around the command language interface and user assistance. The command language interface for the MPE is provided mainly by the IS/1 shell, although VAX/VMS may be used to a lesser extent. In general, it will be desirable to avoid the use of the VMS command language system so that a more unified interface is presented to the user. In the near term, the flexibility of the IS/1 shell should be used to create new commands which directly support and enforce facility-wide policies and standards. User assistance within the MPE is provided mainly through online documentation with tools for rapidly searching through this documentation. During the MPE development, this documentation should be extended to cover the non-IS/1 tools and other MPF specific material such as usage standards.

MPE phase in. One of the major concerns for the MPE, particularly in the near term, has been the achievement of high payoffs from the MPE usage. The determination of the cost impact of the MPF on DMA is extremely difficult, however. A major factor in the near term cost

savings is the plan for the phased introduction of the MPE into the DMA environment. Hence, careful attention should be paid to the development of this plan during the MPE development effort. Benefits should be projected based on better data gained from the use of the near-term experimental system.

H.5.3 METHODOLOGY DEVELOPMENT CONCERNS (PHASE I/IA)

Separation of methodology and facility development. The separation of the facility development task and the methodology development task is highly recommended during the phase I/IA MPE development effort. As these tasks are largely independent of each other and require different areas of expertise, they represent a logical division of effort. Their separation would preclude the compromising of one task due to time or manpower shortages in the other. As both these tasks are essential to the initial success of the MPE, such a separation will lower the overall risk of the initial development effort.

Life cycle coverage. An important concern which impacts the overall effectiveness of the MPE is the tool coverage of the software system life cycle. The proposed near term MPE tool set was selected based on an informal life cycle model. The tool set selected covers most of the activities identified in that life cycle. During methodology development task, however, additional life cycle activities may be identified which are not directly covered by any tool in the proposed tool set. Two such activities which were identified through comparison with the TSD Methodology and which may require MPE support in the long term are formal problem definition and software system design. The formal problem definition activity involves the specification of the function to be performed by a program in a manner which is independent of that program's design. The inclusion of a tool to support the specification and analysis of a problem definition will aid the identification and correction of errors and ambiguities in the specification before they are passed on to the software design. The software system design activity deals with the specification of multiple programs, databases, and interface file structures. The need for a tool to provide such system design support within DMA is likely to grow in the future.

H.5.4 AREAS FOR RESEARCH AND DEVELOPMENT

Determination of evolving needs. The MPE Study attempts to determine needs based on an averaging of the needs stated by DMA personnel regardless of differences in individual training or level of sophistication. Hence, the needs identified are limited to the general case and do not reflect the special needs of individual organizations. In the long term, the perceived needs of DMA personnel will change as they become more sophisticated through tool use, education, and exposure to new technology. In addition, we envision an increased demand for specialization of the facility to meet the needs of particular organizations, projects, or individuals. Procedures for determining these evolving needs and changing the MPE to meet them need to be addressed in future MPE planning and development efforts.

Effect of technology advances. A specific area related to needs determination is the effect of foreseeable technology advances on the MPE. The current needs have been addressed solely from the point of view of the current DMA technological environment. An attempt to determine and address the needs which will develop with the advent of new technology should be made. Specific areas for investigation include the introduction of artificial intelligence into DMA applications, the development of geographic database systems, the use of problem oriented languages, the proliferation of personal computing with high degrees of distribution, and the introduction of new peripheral devices.

MPE evolution. In order to be cost effective in the long run, the MPE must be capable of evolving to meet the changing needs of DMA and to deal with the effects of new technology. Particular concerns in the evolutionary process are the maintenance and enhancement of existing tools, the acquisition of new tools, and the ongoing integration of the tool set. The potential for evolution already exists in the proposed MPE, as the underlying system for the MPE (VAX/VMS, IS/1) is one for which a large number of tools are available or under development. In order to take advantage of this potential, an assessment should be made of long term MPE architectures which will foster smooth growth and evolution, as well as the technical and managerial procedures needed to carry it out.

Management support. One area in which additional tool support may be required in the long term is project management. One aspect of this support is the integration of the programmer and management tools. Currently, VUE does not gather data from any programmer tool, and hence the identification of project status or scheduling updates cannot be done automatically. Integration of these tools would provide management with a better view of the project status and also aid in planning. Another area requiring study is the method for controlling information access and modification within a project. Lastly, procedures and tools for verifying and enforcing compliance with an MPE methodology should be developed.

Multiple environments. The MPE presents a single environment which has been specialized to the task of software development and maintenance within DMA. While a single environment system is sufficient for current DMA needs, there are several advantages to a multiple environment scheme which can aid in the long term usage of the MPE. These advantages include the ability to specialize individual user or project views to meet specific needs, the control of access to project information, the control of additions and modifications for project data, and the ability to provide a smooth, phased evolution of the system tailored to the needs of individual projects.

Project database. Currently, the concept of a project database does not exist in the MPE design, which is in sharp contrast to other facility development efforts such as SREM at TRW. Each individual tool in the MPE maintains its own separate data within the VMS file system, which can result in a fragmentation of the data associated with each project and makes the global analysis of this data more difficult. The organization of the MPE tools around a central project database would be a major step toward unifying the project data collection and integrating the tool set. Particular advantages of this organization are the ability to perform

sophisticated query processing on the entire project database, the establishment of a uniform access structure for the data, and the associated capability of creating tools which collect, analyze, and compare data from many different parts of the project (such as in requirements tracing).

Portability and vendor independence. A final concern which should be assessed is the potential for achieving full portability and vendor independence in the MPE. Portability and vendor independence can be achieved through the use of system architectures and individual components which do not rely on a specific hardware or software support system. Individual components of the system can be made portable by developing them in a high level language which is highly portable, such as C or Ada. Such languages rely on a library of standard support routines which can be redefined easily for each target environment. An entire system can be made independent of vendor-supplied support systems through use of a structure in which all accesses to the support systems (such as a commercial database system) are filtered through a standard interface (as in the TSD System design presented in Section 4) which can be rewritten for each vendor package without affecting the remainder of the system. The applicability of such techniques to the far term MPE system development should be assessed in the R&D task.

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence (C³I) activities. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.

DA